Book Title
Book Editors
IOS Press, 2003

A BDD-like Implementation of an Automata Package

[1]

Jean-Michel Couvreur
LaBRI, Université Bordeaux 1, Talence, France
e-mail: couvreur@labri.fr

Abstract. In this paper we propose a new data structure, called shared automata, for representing deterministic finite automata (DFA). Shared automata admit a strong canonical form for DFA similarly to Binary Decision Diagrams (BDDs). As a result, checking whether two DFAs are equal is a constant-time comparison. A hash-based cache can be used to improve significantly the performance of automata operations. The key points of this structure are the decomposition of the DFA into its strongly connected components and an incremental algorithm based on this decomposition for transforming any DFA into a shared automaton. We experimentally compare PresTaf, a direct implementation of the Presburger arithmetic built on a shared automata package, and the Presburger package LASH based on standard automata algorithms. Experimental results show the great benefit of the new canonical data structure applied to symbolic state space exploration of infinite systems.

Keywords. shared automaton, Presburger arithmetic, symbolic verification, infinite system, Petri net

1. Introduction

Finite Automata are basic structures that appear in many areas of Computer Science and other disciplines. The emergence of new tools based on automata for the manipulation of infinite structures [6,10,1,3] makes improving the efficiency of automata packages a crucial challenge. The present work is motivated by model-checking problems. where most of the algorithms are based on fixed point computations that share many identical sub-computations. A promising track is the realisation of a BDD-like package because BDDs have proved these capability to take advantage of this aspect when using cache technique. Since Bryant's original publication of BDD algorithms [4], there has been a great deal of research in the area. One of the most powerful applications of BDDs has been symbolic model checking, used to formally verify digital circuits and other finite state systems [5]. A BDD package is based on an efficient implementation of the if-then-else (ITE) operator. It uses essentially two principles: a hash table, called unique table, maintains a strong canonical form in BDDs and stores a forest of BDDs sharing common substructures, a hash table, called computed cache, keeps sub-results when evaluating a recursive ITE operation.

[1] A prior version of this work was published in CIAA 2004 as a poster paper of 2 pages. Because a 2 page abstract is sort of a no-op and wish to give access to this work, I hope that this paper will be considered as a normal submission.

Applying the BDD principle to automata is not that easy. Thus a solution to our problem has to design new principles to overcome the following difficulties: define a strong canonical form for automata, handle a forest of automata sharing common substructures, design a constant time procedure to check automata equality and an efficient hash function. Notice that classic notion of minimal automata are far from solving these problems. One needs to design a new structure, well-adapted to substructures sharing and a new algorithm transforming an automaton into this new structure, guaranteeing a strong canonical form.

In this paper we propose a data structure, called shared automata, for representing deterministic finite automata. Informally, a shared automaton codes a strongly connected component of an automaton and its exit states. Thus, an automaton can be considered as an acyclic graph of shared automata. This representation is well-adapted to substructure sharing between automata. We have designed an incremental algorithm based on this decomposition producing shared automata where states respect some canonical order. During the canonisation of an automaton, produced shared automata are stored in a unique table, guaranteeing a strong canonical form like for BDDs. During this evaluation, sub-results are stored in a computed cache, avoiding unnecessary re-evaluation of sub-expressions. In our system, automata operations, as set operations, are obtained on-the-fly : applying the canonical algorithm on a computed non canonical representation of the result. We experimentally compare PresTaf, a direct implementation of the Presburger arithmetic built on the shared automata package, and the Presburger package LASH [10] based on standard automata algorithms. The goal of this experimentation is to evaluate the benefits that shared automata techniques can bring to systems using standard automata algorithms. Comparison with other kind of Presburger package is out of the scope of our experimentation. We chose a classic problem verification: the backward symbolic state space exploration for Petri nets. Experimental results show the great benefit of the new canonical structure applied to this kind of problems. As BDD [11], the main factor of this benefit is the computed cache. Indeed, the iterations of a state space exploration share many sub-problems.

The rest of the paper is organised as follows. We first present *exit automata*, an extension of automata used to represent SCCs of automata. Then, we give a formal definition of *shared automata* and state some sufficient conditions for the canonicity of a forest of shared automata. The next section is dedicated to the implementation and focuses on the algorithm to transform automata into shared automata. Finally, we give some experimental results.

2. Deterministic Finite Exit Automata

Definition 1 *A* deterministic finite exit automaton *(DFEA) over the finite alphabet Σ is a structure $\mathcal{O} = \langle Q, E, \Sigma, \delta, F\rangle$ where Q and E are finite disjoint sets of states, $F \subseteq Q$, and the transition function is given by $\delta : Q \times \Sigma \to Q \cup E$. States of Q are called* local states*, and states of E are called* exit states*. A DFEA with an initial state in Q, $\langle \mathcal{O}, q\rangle$, denoted $\mathcal{O}_q$, is called a* marked DFEA.

When E is empty, the automaton is just a deterministic finite automaton (DFA). The transition function δ extends in a partial function on words:

$$\begin{array}{ll} \delta^* : (Q \cup E) \times \Sigma^* \to Q \cup E & \\ \delta^*(p, \epsilon) & = p \\ \delta^*(p, xa) & = \begin{cases} \delta(\delta^*(p, x), a) & \text{if } \delta^*(p, x) \in Q \\ \text{undefined} & \text{otherwise} \end{cases} \end{array}$$

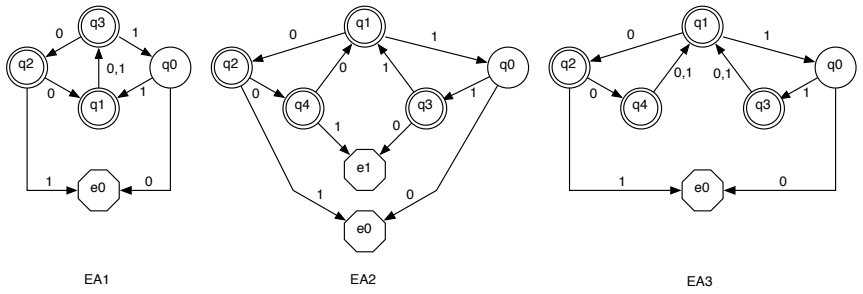


Figure 1. Examples of exit automata

where x is a word, and $a \in \Sigma$. Thus a marked DFEA *locally accepts* input x on local state q if $\delta^*(q, x) \in F$; we denote the set of locally accepted words $L(\mathcal{O}_q)$. Note that when the exit state set E is empty, a DFEA is a DFA and locally-accepted words are accepted words. For any exit state u, we say that x is a *prefix word* of u from q if $\delta^*(q, x) = u$; the set of prefix words of u is denoted $L[u](\mathcal{O}_q)$.

Nerode equivalence is extended to (local and exit) states of DFEA; two states are equivalent if they locally accept the same language and have the same prefix language for any exit state: $p \sim q \Leftrightarrow (L(\mathcal{O}_p) = L(\mathcal{O}_q) \wedge \forall u \in E : L[u](\mathcal{O}_p) = L[u](\mathcal{O}_q))$. Nerode equivalence remains compatible with respect to the transition function. Thus the quotient DFEA can be defined as $\mathcal{O}_{/\sim} = \langle Q_{/\sim}, E, \Sigma, \delta_{/\sim}, F_{/\sim} \rangle$ with $\delta_{/\sim}(p_{/\sim}, a) = \delta(p, a)_{/\sim}$. Note that $L(\mathcal{O}_{/\sim_{q_{/\sim}}}) = L(\mathcal{O}_q)$, and $L[u](\mathcal{O}_{/\sim_{q_{/\sim}}}) = L[u](\mathcal{O}_q)$ for any local state q. A DFEA is said to be *minimal* if no two distinct local states are equivalent. Minimizing a DFEA still consists in computing the quotient automaton $\mathcal{O}_{/\sim}$. The notion of DFEA extends the definition of DFA, but does not immediately provide an interesting structure in our study. We will consider strongly connected DFEA.

Definition 2 *A DFEA $\mathcal{O} = \langle Q, E, \Sigma, \delta, F \rangle$ is a* strongly connected DFEA *(SC-DFEA) if any local state is reachable from any local state: $\forall p, q \in Q, \exists x \in \Sigma^* : \delta^*(p, x) = q$.*

Example 1 *Figure 1 gives three examples of exit automata. While $EA1$ and $EA2$ are minimal, minimizing $EA3$ leads to $EA1$ (i.e. $q3 \sim q4$). Notice that when minimizing a SC-DFEA, the quotient automaton remains a SC-DFEA.*

3. Shared Automata

Shared automata represent the strongly connected components of an automaton in term of SC-DFEAs and the connections by functions mapping exit states to states of the automaton. Consider a finite alphabet Σ for all automata, shared automata and marked shared automata are inductively defined as follow :

Definition 3 Shared Automata *and* marked shared automata *are both inductively defined by:*

- *if $\mathcal{O}$ is an SC-DFEA with no exit state then $\mathcal{O}$ is a shared automaton,*
- *if $\mathcal{O}$ is an SC-DFEA with no exit state and q a local state of $\mathcal{O}$ then $\mathcal{O}_q$ is a marked shared automaton,*

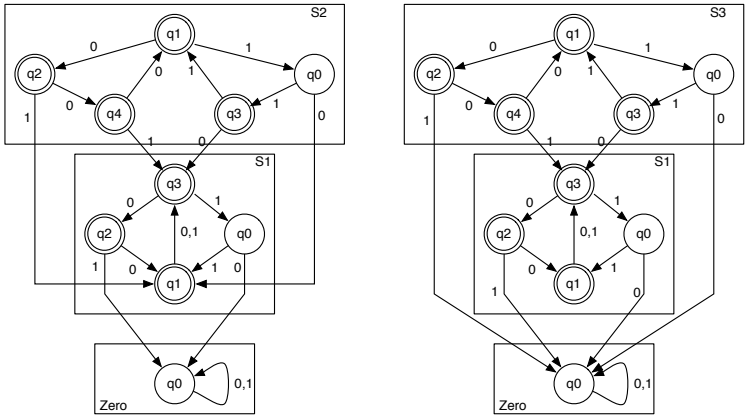


Figure 2. Decomposition of two automata into shared automata

- *if $\mathcal{O}$ is an SC-DFEA, and λ a mapping associating to any exit state of $\mathcal{O}$, a marked shared automaton, then $\mathcal{O}(\lambda)$ is a shared automaton,*
- *if $\mathcal{O}(\lambda)$ is a shared automaton, and q is a local state of $\mathcal{O}$, then $\mathcal{O}_q(\lambda)$ is a marked shared automaton.*

Functions λ are called *bind functions*. Note that a marked shared automaton can be also defined as a marked SC-DFEA associated with a bind function. When a SC-DFEA $\mathcal{O}$ has no exit, we use the convention $\mathcal{O}(\emptyset)$ for representing the shared automaton $\mathcal{O}$. Thus any shared automaton has the form $\mathcal{O}(\lambda)$. We naturally extend the convention for marked shared automata with no exit: a marked shared automaton $\mathcal{O}_q$ is denoted by $\mathcal{O}_q(\emptyset)$. A marked shared automaton $\mathcal{O}_q(\lambda)$ can be written $\mathcal{S}_q$ where $\mathcal{S} = \mathcal{O}(\lambda)$. Sometimes, marked shared automata will be considered as shared automata: for instance, we may say "the shared automaton $\lambda(u)$".

Let gives some basic notions and a formal definition of the DFA representing a shared automaton.

Definition 4 *The* component set *of a shared automaton $\mathcal{S} = \mathcal{O}(\lambda)$, written SCC($\mathcal{S}$), is a set of shared automata such that*

$$SCC(\mathcal{S}) = \{\mathcal{S}\} \cup \bigcup_{u \in E} SCC(\lambda(u)),$$

where E is the exit set of $\mathcal{O}$, and $\lambda(u)$ are considered as shared automata.

Definition 5 *The* depth *of a shared automaton $\mathcal{S} = \mathcal{O}(\lambda)$, written DEPTH($\mathcal{S}$), is defined as:*

$$DEPTH(\mathcal{S}) = \begin{cases} 0 & \text{if } \lambda = \emptyset \\ 1 + \max_{u \in E} DEPTH(\lambda(u)) & \text{otherwise} \end{cases}$$

Definition 6 *The* DFA *represented by a shared automaton S, denoted DFA($\mathcal{S}$), is defined by :*

- *States are marked shared automata $\mathcal{O}_q(\lambda)$ where $\mathcal{O}(\lambda)$ is a shared automaton of SCC($\mathcal{S}$) and q is a local state of $\mathcal{O}$,*

- *State $\mathcal{O}_q(\lambda)$ is final if q is final in $\mathcal{O}$,*
- *Transition function δ is given by :*

$$\delta(\mathcal{O}_q(\lambda), a) = \begin{cases} \mathcal{O}_{\delta(q,a)}(\lambda) & \text{if } \delta(q,a) \text{ is a local state} \\ \lambda(\delta(q,a)) & \text{otherwise} \end{cases}$$

Proposition 1 *Any DFA is isomorphic to the DFA of a shared automaton.*

Proof. For each SCC of a DFA, one can build an exit automaton : local states are states of the SCC, exit states are states connected to SCC states not in the SCC and the transition function is the restriction to the SCC of transition function of DFA. Then for each SCC, we consider the shared automaton composed with its exit automaton and a bind function connecting each exit state to its corresponding state as a local state of a shared automaton associated to its SCC. □

Example 2 *Figure 2 gives actually the representation of four shared automata:* Zero, S1, S2, S3. Zero *is an exit automaton with no exit state, then a shared automaton. The others are defined from the exit automata $\mathcal{O}1$ and $\mathcal{O}2$ (see Figure 1):*

- $\mathcal{S}1 = \mathcal{O}1(\lambda_1)$ *with* $\lambda_1(e0) = Zero_{q0}$,
- $\mathcal{S}2 = \mathcal{O}2(\lambda_2)$ *with* $\lambda_2(e0) = \mathcal{S}1_{q1}$, $\lambda_2(e1) = \mathcal{S}1_{q3}$,
- $\mathcal{S}3 = \mathcal{O}2(\lambda_3)$ *with* $\lambda_3(e0) = Zero_{q0}$, $\lambda_3(e1) = \mathcal{S}1_{q3}$.

Note that $SCC(Zero) = \{Zero\}$, $DEPTH(Zero) = 0$, $SCC(\mathcal{S}1) = \{Zero, \mathcal{S}1\}$, $DEPTH(\mathcal{S}1) = 1$, *etc.*

The canonicity problem of shared automata representation requires a preliminary study about local characterisation of minimal automata. First, we give the definition of a minimal shared automaton. Second, we state necessary conditions for the minimality property. Then we give sufficient conditions which lead to a strong canonical representation of shared automata.

Definition 7 *A shared automaton is said to be* minimal *if its DFA is minimal.*

Proposition 2 *Let $\mathcal{S}$ be a shared automaton. Let $\mathcal{O}(\lambda)$ be a component of $\mathcal{S}$. If $\mathcal{S}$ is minimal, then $\mathcal{O}$ is minimal.*

Proof. This property is based on the fact that for every local state p of $\mathcal{S}$:

$$L(\mathcal{S}_p) = L(\mathcal{O}_p) \cup \bigcup_{u \in E} L[u](\mathcal{O}_p) \cdot L(\lambda(u))$$

where E is the exit state set of $\mathcal{O}$. Then, if $\mathcal{O}$ is not mininal, $\mathcal{S}$ is not minimal. □

The second condition is based on the notion of shared automaton homomorphisms. Informally, an homomorphism is a mapping between states of two shared automata, compatible with the transition functions.

Definition 8 *Let $\mathcal{S} = \mathcal{O}(\lambda)$, $\mathcal{S}' = \mathcal{O}'(\lambda')$ be two shared automata. Denote $\mathcal{O} = \langle Q, E, \Sigma, \delta, F\rangle$ and $\mathcal{O}' = \langle Q', E', \Sigma, \delta', F'\rangle$. Let $h : Q \cup E \to Q' \cup E'$ be mapping which fulfils $h(Q) \subseteq Q'$, $h(F) \subseteq F'$ and $h(Q \setminus F) \subseteq Q' \setminus F'$. Then h is an homomorphism from $\mathcal{S}$ to $\mathcal{S}'$ if for any pair (p, a) in $Q \times \Sigma$ where $q = \delta(p, a)$ and $q' = \delta'(h(p), a)$, we have the following conditions:*

- *if $q \in Q$, then $q' \in Q'$ and $h(q) = q'$,*
- *if $q \in E$ and $q' \in E'$, then $\lambda(q) = \lambda'(q')$,*
- *if $q \in E$ and $q' \in Q'$, then $\lambda(q) = \mathcal{S}'_{q'}$.*

We say that $\mathcal{O}$ is homomorphic to $\mathcal{O}'$ if there exists an homomorphism $h : \mathcal{O} \to \mathcal{O}'$. Moreover, if h is bijective, $\mathcal{O}$ and $\mathcal{O}'$ are said to be isomorphic.

The following proposition states that homomorphism links marked shared automata accepting the same words.

Proposition 3 *Let $\mathcal{S} = \mathcal{O}(\lambda)$, $\mathcal{S}' = \mathcal{O}'(\lambda')$ be two shared automaton. Let $h : \mathcal{S} \to \mathcal{S}'$ be an homomorphism. Let p a local state of $\mathcal{O}$. Then $L(\mathcal{S}_p) = L(\mathcal{S}'_{h(p)})$.*

Proof. Let $x = a_1 \cdots a_n$ be a word in Σ^*. If for all prefixes $x_i = a_1 \cdots a_i$, $\delta(p, x_i)$ are defined and are local states of $\mathcal{O}$, $\delta'(h(p), x_i)$ are local states of $\mathcal{O}'$. From the definition of an homomorphism, both target states $\delta(p, x)$, $\delta(h(p), x)$ are final or not final, and then $x \in L(\mathcal{S}_p) \Leftrightarrow x \in L(\mathcal{S}'_{h(p)})$. If for a prefix x_i, $\delta(p, x_i)$ is an exit state, then $x \in x_i \cdot L(\lambda(\delta(p, x_i)))$. From the definition of an homomorphism, if $\delta'(h(p), x_i) \in E'$, then $\lambda(\delta(p, x_i)) = \lambda'(\delta'(h(p), x_i))$, otherwise $\lambda(\delta(p, x_i)) = \mathcal{S}'_{\delta'(h(p), x_i)}$. For both cases, we deduce that $x \in L(\mathcal{S}_p) \Leftrightarrow x \in L(\mathcal{S}'_{h(p)})$. □

Then, we deduce the second necessary condition for the minimality of shared automata.

Corollary 1 *Let $\mathcal{S}$ be a shared automaton. If $\mathcal{S}$ is minimal, then the shared automata in SCC($\mathcal{S}$) are pairwise non homomorphic.*

Example 3 *Shared automaton $S3$ in Figure 2 is not minimal, instead its exit automata are minimal. Indeed, $S3$ is homomorphic to $S1$: $h(q0) = q0$, $h(q1) = q3$, $h(q2) = q2$, $h(q3) = h(q4) = q1$, $h(e0) = q3$ and $h(e1) = e0$. On the other hand, we can prove that $S2$ is not homomorphic to $S1$ and moreover that $S2$ is minimal.*

The following proposition gives a finer characterisation of homomorphic shared automata. It will be used for the canonical representation of shared automata, and also for the optimisation of our implementation.

Proposition 4 *Let $\mathcal{S} = \mathcal{O}(\lambda)$, $\mathcal{S}' = \mathcal{O}'(\lambda')$ be two shared automata. Denotes E and E' the exit state sets of $\mathcal{O}$ and $\mathcal{O}'$, and Q' the local state set of $\mathcal{O}'$. If $\mathcal{S}$ is homomorphic to $\mathcal{S}'$, then*

$$\lambda(E) \subseteq \lambda(E') \cup \{\mathcal{S}'_{q'} | q' \in Q'\}$$

Moreover if $\lambda(E) \subseteq \lambda(E')$ then $\lambda(E) = \lambda(E')$, otherwise DEPTH($\mathcal{S}$) = DEPTH($\mathcal{S}'$) + 1 and $\forall \mathcal{S''}_q \in \lambda(E)$: DEPTH($\mathcal{S''}$) = DEPTH($\mathcal{S}'$) $\Rightarrow \mathcal{S''} = \mathcal{S}'$.

Proof. Let $h : \mathcal{S} \to \mathcal{S}'$ be an homomorphism. The fact that $\lambda(E) \subseteq \lambda(E') \cup \{\mathcal{S}'_{q'} | q' \in Q'\}$ is an immediat consequence of the definition of an homomorphism. Assume that $\lambda(E) \subseteq \lambda(E')$. Let $e' \in E'$, $p \in Q$ and $x \in \Sigma^*$ such that $e' = \delta'(h(p), x))$. Then $\delta(p, x) \in E$ and $h(\delta(p, x)) = e'$. Indeed if $\delta(p, x)$ is not defined, then there is a prefix y of x such that $h(\delta(p, y))$ is in Q'. This is in contradiction with $\lambda(E) \subseteq \lambda(E')$; if $\delta(p, x)$ is defined, $\delta(p, x)$ is also an exit state, because $h(\delta(p, x))$ is an exit state. Thus, any exit state of $\mathcal{O}'$ is in $h(E)$. Assume that $\lambda(E) \not\subseteq \lambda(E')$.

Then DEPTH($\mathcal{S}$) = DEPTH($\mathcal{S}'$) + 1 and the only marked shared automata in $\lambda(E)$ with depth($\mathcal{S}$') are in $\{\mathcal{S}'_{q'}|q' \in Q'\}$. □

Enforcing the necessary conditions leads to rules to achieve a strong canonical form of marked shared automata. The key point is the notion of structurally minimal set of shared automata.

Definition 9 *Let $\mathcal{F}$ be a set of shared automata. The set $\mathcal{F}$ is* structurally minimal *if all pairs of shared automata $\mathcal{S} = \mathcal{O}(\lambda)$ and $\mathcal{S}'$ fulfil the following conditions :*

1. *The bind function λ of $\mathcal{S}$ is injective or empty.*
2. *Let E be the exit state set of $\mathcal{O}$. $\mathcal{S}$ is not homomorphic to any shared automata of $\lambda(E)$.*
3. *The exit automaton $\mathcal{O}$ is minimal.*
4. *If $\mathcal{S}$ and $\mathcal{S}'$ are isomorphic then $\mathcal{S} = \mathcal{S}'$.*

Finally, we conclude with the proposition laying down the rules for obtaining a canonical representation of shared automata.

Proposition 5 *Let $\mathcal{S}_q$, $\mathcal{S}'_{q'}$ be two shared automata. If SCC($\mathcal{S}$) ∪ SCC($\mathcal{S}'$) is structurally minimal, then the two following properties are equivalent :*

1. $\mathcal{S} = \mathcal{S}'$ *and* $p = p'$,
2. $L(\mathcal{S}_p) = L({\mathcal{S}'}_{p'})$

Proof. Suppose that there exists $\mathcal{S}_p \neq {\mathcal{S}'}_{p'}$ with $L(\mathcal{S}_p) = L({\mathcal{S}'}_{p'})$. Assume that $\mathcal{S}_p$, ${\mathcal{S}'}_{p'}$ is selected such that DEPTH($\mathcal{S}$) + DEPTH($\mathcal{S}'$) is minimal. Let consider the two cases $\mathcal{S} = \mathcal{S}'$ and $\mathcal{S} \neq \mathcal{S}'$. Denote $\mathcal{S} = \mathcal{O}(\lambda)$ and $\mathcal{S}' = \mathcal{O}'(\lambda')$.

(1) If $\mathcal{S} = \mathcal{S}'$. Firstly, $L(\mathcal{O}_p) \subseteq L(\mathcal{O}_{p'})$. Let $x \in L(\mathcal{O}_p)$. If $\delta(p', x)$ is local state, then $\delta(p', x)$ is final, because $x \in L(\mathcal{S}_p) = L({\mathcal{S}'}_{p'})$; if $\delta(p', x)$ is not defined or is an exit state, there exists a prefix y of x such that $\delta(p', y)$ is an exit state, and then $\mathcal{S}_\delta(p, y) \neq \lambda(\delta(p', y))$ and $L(\mathcal{S}_\delta(p, x)) = L(\lambda(\delta(p', y)))$. This is in contradiction with the minimality of DEPTH($\mathcal{S}$) + DEPTH($\mathcal{S}'$). For the same raison $L(\mathcal{O}_{p'}) \subseteq L(\mathcal{O}_p)$. Secondly, $L[u](\mathcal{O}_p) \subseteq L[u](\mathcal{O}_{p'})$ for any exit state u. Let $x \in L[u](\mathcal{O}_p)$. If $\delta(p', x)$ is a local state then $\lambda(u) \neq \mathcal{S}_\delta(p', x)$ and $L(\lambda(u)) \neq L(\mathcal{S}_\delta(p', x))$. This is in contradiction with the minimality of DEPTH($\mathcal{S}$) + DEPTH($\mathcal{S}'$). If $\delta(p', x)$ is an exit state v not equal to u; if $\lambda[u] \neq \lambda[v]$, then these two marked shared automata contradict the minimality of DEPTH($\mathcal{S}$) + DEPTH($\mathcal{S}'$), otherwise if $\lambda[u] = \lambda[v]$, λ is not injective and then SCC($\mathcal{S}$) ∪ SCC($\mathcal{S}'$) is not structurally minimal. If $\delta(p', x)$ is not defined, there exists a prefix y of x such that $\delta(p', y)$ is an exit state, and then $\mathcal{S}_\delta(p, y)$, $\lambda(\delta(p', y))$ contradict the minimality of DEPTH($\mathcal{S}$) + DEPTH($\mathcal{S}'$). Since $x \in L[u](\mathcal{O}_{p'})$. For the same raison $L[u](\mathcal{O}_{p'}) \subseteq L[u](\mathcal{O}_p)$. We have proven that state p and p' are Nerode equivalent, and then $\mathcal{O}$ is not minimal. This is in contradiction with the structurally minimality of SCC($\mathcal{S}$) ∪ SCC($\mathcal{S}'$).

(2) If $\mathcal{S} \neq \mathcal{S}'$ with DEPTH($\mathcal{S}$) ≥ DEPTH($\mathcal{S}'$). Let q be a state of $\mathcal{O}$. Let $L_q = \{x \in \Sigma^* | q = \delta(p, x)\}$. Firstly, $\delta'(p', x)$ is defined for all $x \in L_q$. If $\delta(p', x)$ is not defined, then there exist a prefix y of x such that $\delta(p', y)$ is an exit state. $\mathcal{S}_\delta(p, y) \neq \lambda(\delta(p', y))$ because DEPTH($\mathcal{S}$) ≥ DEPTH($\mathcal{S}'$) > DEPTH($\lambda(\delta(p', y))$), and then $\mathcal{S}_\delta(p, y)$, $\lambda(\delta(p', y))$ contradict the minimality of DEPTH($\mathcal{S}$) + DEPTH($\mathcal{S}'$). Secondly, if q is local then $\delta'(p', x)$ is local for all $x \in L_q$. If $\delta'(p', x)$ is an exit state, for depth argument $\mathcal{S}_\delta(p, x) \neq \lambda(\delta(p', x))$, and then $\mathcal{S}_\delta(p, x)$, $\lambda(\delta(p', x))$ contradict the minimality of DEPTH($\mathcal{S}$) + DEPTH($\mathcal{S}'$). Thirdly,

if $\delta(p,x) = \delta(p,y)$ for all $x,y \in L_q$. If there exists two word $x,y \in L_q$ such that $\delta(p,x) \neq \delta(p,y)$, then $\mathcal{S}'_{\delta(p,x)}$ and $\mathcal{S}'_{\delta(p,x)}$ accept the same language and have minimal depth sum. From the case (1), we conclude with a contradiction. Now, we are in position to define the homomorphism $h : \mathcal{S} \to \mathcal{S}'$ with $h(q) = \delta(p',x)$ where x is any word in L_q. Let E and E' the exit state set of $\mathcal{O}$ and $\mathcal{O}'$. If $\lambda(E) \not\subseteq \lambda'(E')$, then from Proposition 4, $\mathcal{S}'$ is an image of λ and then SCC($\mathcal{S}$) $\cup$ SCC($\mathcal{S}'$) is not structurally minimal. If $\lambda(E) \subseteq \lambda'(E')$, then from Proposition 4, $\lambda(E) = \lambda(E')$ and then DEPTH($\mathcal{S}$) = DEPTH($\mathcal{S}'$). Let prove that h is surjective. Let q' a state of $\mathcal{O}'$, let $x \in \Sigma^*$ such that $q' = \delta'(p',x)$. $\delta(p,x)$ is defined, otherwise there exists a prefix y of x such that $\delta(p,y)$ is an exit state and for depth argument $\lambda(\delta(p,y)) \neq \mathcal{S}'_{\delta'(p',y)}$ and then they contradict the minimality of DEPTH($\mathcal{S}$) + DEPTH($\mathcal{S}'$). Moreover h is injective. Indeed, if two states of $\mathcal{O}$ have the same image, they induce two shared automaton accepting the same language and having minimal depth sum. If the two states are local, from the case (1), we conclude with a contradiction, otherwise the depth sum is less than DEPTH($\mathcal{S}$) + DEPTH($\mathcal{S}'$). We have prove that $\mathcal{S}$ and $\mathcal{S}'$ are isomorphic, and then conclude that SCC($\mathcal{S}$) $\cup$ SCC($\mathcal{S}'$) is not structurally minimal. □

Notice that the structurally minimal property induces minimal shared automata.

Corollary 2 *Let $\mathcal{S}$ be a shared automaton. If SCC($\mathcal{S}$) is structurally minimal then $\mathcal{S}$ is minimal.*

4. Shared Automata Implementation

The goal of this section is to translate the theoretical results of previous section, mainly the notion of structurally minimal set, into an implementation. We borrow the idea of unique table for BDD to convert equality of structures into equality of reference (i.e. pointer). Transforming equality of exit automata modulo a permutation of local states is the main challenge of the implementation. We propose a simple solution to this problem : a light modification of Hopcroft'algorithm [8,7,9].

4.1. Strong Canonical Representation of Marked Shared Automata

Basic structures for shared automaton representations are exit automata (SC-DFEA), shared automata, and marked shared automata. Figure 3 gives the definition of these structures in Java like. Following BDD implementation principles, two hash tables impose a physical canonical form of exit automaton structures, so that (reference) equality of exit automata (and equality of shared automata) are equivalent to the structure equality. Hence, these two hash tables are called *exit automaton unique table* and *shared automaton unique table*. Proposition 5 gives constraints on the unique-tables leading to a strong canonical form for marked shared automata.

We propose to reformulate them to take into account implementation constraints :

1. For all shared automata in the shared automaton unique table, the binding function is injective. Moreover, we impose that the table entries representing the binding function are ordered lexically with respect to the depth of the shared automaton entry, at last with respect to the reference of shared automaton entry, and otherwise with respect to the initial state of the marked shared automaton entry.

```
1  static final int alphabetSize ; // Alphabet = {0 ... alphabetSize−1}
2
3  class ExitAutomaton {
4      int nbLocalStates ; // LocalStates = {0 ... nbLocalState−1}
5      int nbExitStates ; // ExitStates = {−1 ... − nbOutState}
6      int [ ][ ] succ ; // transition function
7      boolean [ ] isFinal ; // final state caracteristic function
8  }
9
10 class SharedAutomaton {
11     ExitAutomaton exitAutomaton;
12     MarkedSharedAutomaton [ ] bindFunction ; // image of exit state −k is bindFunction[k−1]
13     int depth;
14 }
15
16 class MarkedSharedAutomaton {
17     SharedAutomaton sharedAutomaton;
18     int initial ; // initial < sharedAutomaton.exitAutomaton.nbLocalStates
19 }
```

Figure 3. Basic structures for shared automaton representations

2. For all shared automata in the shared automaton unique table, it is not homomorphic with any entry of the bind function (i.e given any entry of maximal depth of the shared automaton, it is not homomorphic with this entry).
3. All exit automata (in the exit automaton unique table) are minimal.
4. If two exit automata (in the exit automaton unique table) are isomorphic up to a permutation of the local states, then they are equal.

Note that from the theory to the practice, we add an additional constraint for Condition 1: ordering the bind function image. This constraint is easy to preserve, and gives two additional properties. First, when checking condition 2, finding an entry with maximal depth can be done within constant time. Second, condition 4 has been reduced to a condition on exit automaton unique table: Condition 1 assumes that the bind functions of two isomorphic shared automata are equal, so isomorphism is reduced to equality up to a permutation of these local state sets.

Example 4 *Tables 1 and 2 give the contents of the two unique tables after the canonisation of the shared automaton $\mathcal{S}2$ (see Figure 2). Notice that a marked shared automaton $\mathcal{S}_q$ is simply denoted by a pair $(\mathcal{S}, q)$. In our implementation, we have defined two constants: $Zero = \mathcal{S}_{Zero}$ and $One = \mathcal{S}_{One}$. They represent respectively the automata recognizing no word and all words. Note that* `bindFunction` *arrays are correctly ordered. However, the choice of the local state ordering will be justified in the next section and is the result of our adapted Hopcroft algorithm.*

4.2. Transforming DFA into Shared Automata

Transforming DFA into shared automata is the central point of the method. The goal is to compute and store a marked shared automaton having the same language as the given marked automaton, and to ensure the integrity constraints of the unique table. The algorithm (see Figure 4) is based on local evaluations of the strongly connected components (SCC) starting with

Réf.	nbLocalStates	nbOuputStates	succ	isFinal
$\mathcal{O}_{Zero}$	1	0	[[0,0]]	[0]
$\mathcal{O}_{One}$	1	0	[[0,0]]	[1]
$\mathcal{O}1$	4	1	[[-1,1],[3,3],[1,-1],[2,0]]	[0,1,1,1]
$\mathcal{O}2$	5	2	[[-1,3],[2,0],[4,-1],[-2,1],[1,-2]]	[0,1,1,1,1]

Table 1. Exit automaton unique table

Ref	exitAutomaton	bindFunction	depth
$\mathcal{S}_{Zero}$	$\mathcal{O}_{Zero}$	[]	0
$\mathcal{S}_{One}$	$\mathcal{O}_{One}$	[]	0
$\mathcal{S}1$	$\mathcal{O}1$	$[Zero]$	1
$\mathcal{S}2$	$\mathcal{O}2$	$[(\mathcal{S}1, 1), (\mathcal{S}1, 3)]$	2

Table 2. Shared automaton unique table

the deepest one, and on the storage of all intermediate results in a computation cache. The processing of a component is carried out in three steps.

1. The first step consists of calculating a representation of the SCC in terms of shared automata. Let us note that the values of the entries of the bind function are deduced from smaller depth SCC evaluations, and then are retrieved from the computation cache. The requirements of the first integrity constraint are fulfilled while ensuring that entries are pairwise distinct and correctly ordered.
2. The goal of the second step (see Ligne 12 in Figure 4) is to test whether a shared automaton is homomorphic with one of these entries, and more particularly one with a greater depth entry. If so, the homomorphism maps any state of the SCC to an equivalent marked shared automaton. Otherwise, the computation must precede on the last step.
3. The last step (see Ligne 20 in Figure 4) consists in minimizing the exit automaton in a strongly canonical form, and then to ensure the two last integrity constraints.

The first step consists in a SCC traversal computation and the transformation of SCCs into non-canonical shared automata are easy to implement. Note however that it is useless to continue the computation beyond states whose result is in the cache or corresponds to a state of a shared automaton already in the unique table. This technique will play a major role for the efficiency of our method.

The algorithm carried out in step 2 is quite simple: once having selected a local state p of the shared automaton and a suitable entry of the bind function, one has to check all local states q of the selected entry to see whether there exists an homomorphism binding state p to state q. This test consists in traversing the shared automaton from state p, calculating new values of the homomorphism while checking the coherence of the results. Note that the complexity of this algorithm is in $O(\Sigma) \cdot O(n) \cdot O(n')$ where n, n' denote the number of local states of the shared automaton and its selected entry. (`homomorphicCheck`)

The implementation of step 3 (see Figure 5) is based on a light modification of Hopcroft's algorithm for minimizing automata [8,7,9]. Our adapted Hopcroft algorithm (see Figure 6) is extended to manage exit automata. This goal is carried out in lines 25-27: not only do the final states induce an initial equivalence class partition, the predecessor sets of each exit states require new refinements of the partition. For the remainder, the algorithm is almost the exact

```
1  HashMap cache = new HashMap();
2
3  MarkedSharedAutomaton canonical(MarkedAutomaton A){
4      for all SCC of A in a topologic order {
5          SharedAutomaton shared;
6          MarkedAutomaton [ ] f;
7
8          Assign to "shared" be a (non canonical) shared automaton representation of SCC
9          Assign to "f" be the isomorphism linking local states of "shared" to states of SCC
10
11         // check if shared is homomorphic to an image of the bind function
12         MarkedSharedAutomaton [ ] g = homomorphicCheck(shared);
13         if (g != null) {
14             for all "p" local state of "shared" {
15                 cache[f[p ]] = g[p];
16             }
17         }
18         else {
19             // compute a canonical represention of the exit automaton
20             ExitAutomaton,int [ ] < exitAutomaton,h> = minimize(shared.exitAutomaton);
21
22             SharedAutomaton newShared
23                 = unique(new SharedAutomaton(exitAutomaton,shared.bindFunction));
24             for all "p" local state of "shared" {
25                 cache[f[p ]] = new markedSharedAutomaton(newShared,h[p]);
26             }
27         }
28     }
29     return cache[A];
30 }
```

Figure 4. Transforming a marked automaton into shared automata

copy of the original algorithm with Gries's improvement. However, we add a sort instruction of the set $pivot$ in line 9 to force the algorithm to procede independently of the initial order of local states. The order of the steps of refinement of classes is not important to the validity of the algorithm. Within our framework, this modification produces benefits, as when the algorithm is applied to a minimal exit automaton, local states are ordered independently to their initial order. As a result, applying our adapted Hopcroft algorithm twice (see Ligne 2 and 12 of Figure 5) produces a strong canonical minimal automaton. We did not go through the fine data structures leading to a $O(|\Sigma|) \cdot O(n \cdot \lg n)$ time complexity for classical algorithm [7,9]; however, we can prove that our modification does not increase the time complexity.

Assuming that hash tables are perfect, the time complexity of the transformation algorithm is $O(|\Sigma|) \cdot O(scc) \cdot O(n)$ where scc is the number of states of the largest SCC. Note that when the sizes of the components remain bounded (for instance, acyclic automata), the algorithm has a linear time complexity. In theory, perfect hash table is ideal, in practice one may prefer tree map structures to reduce the worst case time complexity. Informally, we obtain an overhead for table management of the form $O(|\Sigma|) \cdot O(n) \cdot O(\lg(n + K))$ where K indicates a constant describing the size of the data stored in unique tables.

Example 5 *Figure 7 gives the representation of two (non canonical) shared automata for the*

```
1  ExitAutomaton,int [ ] minimize(ExitAutomaton exitAutomaton){
2      ExitAutomaton,int [ ] < minimal,h> = adaptedHopcroft(exitAutomaton);
3
4      // if "exitAutomaton" is minimal then "minimal" is the canonical form
5      if (minimal.nbLocalState == exitAutomaton.nbLocalState ) {
6          return <minimal,h>;
7      }
8
9      else { // otherwise "minimal" is minimal, and then we compute a canonical form
10         int [ ] g;
11         int [ exitAutomaton.nbLocalState ] res ;
12         <minimal,g> = adaptedHopcroft(minimal);
13         for all "i" local state of "exitAutomaton" {
14             res[ i ] = g[h[ i ]];
15         }
16         return <minimal,res >;
17     }
18 }
```

Figure 5. Canonical algorithm for exit automata

evaluation of the intersection and the union of shared automata $(\mathcal{S}1, 3)$ *and* $(\mathcal{S}2, 1)$*. Note that we stop the evalution on states identified as exit states. We simply use the well know assertions:* $A \cap \emptyset = \emptyset$*,* $A \cap A = \emptyset$*,* $A \cup \emptyset = A$ *and* $A \cup A = A$*. For this example, we use simple assertions to stop the evaluation. From the implementation point of view, there are integrated in the definition of the (non canonical) marked automaton representing the result of the operation. Another way to stop the evaluation is to retrieve the resulting sub-expression from the computed cache.*

The canonical algorithm applied to the intersection automaton stops with the detection of an homomorphism. Indeed, this shared automaton is exactly automaton $\mathcal{S}3$ *(see Figure 2). Then, before returning the result* $(\mathcal{S}3, 1)$*, the algorithm stores in the cache table the following sub-results:* $(\mathcal{S}1, 3) \cap (\mathcal{S}2, 1) = (\mathcal{S}3, 1)$*,* $(\mathcal{S}1, 2) \cap (\mathcal{S}2, 2) = (\mathcal{S}3, 2)$*,* $(\mathcal{S}1, 1) \cap (\mathcal{S}2, 4) = (\mathcal{S}3, 4)$*,* $(\mathcal{S}1, 0) \cap (\mathcal{S}2, 0) = (\mathcal{S}3, 0)$*,* $(\mathcal{S}1, 1) \cap (\mathcal{S}2, 3) = (\mathcal{S}3, 3)$*.*

The canonical algorithm applied to the union automaton leads the minimisation of the exit automaton. On this example, the adapted Hopcroft is applied only once, because the shared automaton is already minimal. However, its application has ordered the local states in a canonical form. Note that the resulting exit automaton is exactly $\mathcal{O}1$ *and then its canonical value will be recoved from the exit automaton unique table. Thus, before returning the result* $(\mathcal{S}2, 1)$*, the algorithm stores in the cache table the following sub-results:* $(\mathcal{S}1, 3) \cup (\mathcal{S}2, 1) = (\mathcal{S}2, 1)$*,* $(\mathcal{S}1, 2) \cup (\mathcal{S}2, 2) = (\mathcal{S}2, 2)$*,* $(\mathcal{S}1, 1) \cup (\mathcal{S}2, 4) = (\mathcal{S}2, 4)$*,* $(\mathcal{S}1, 0) \cup (\mathcal{S}2, 0) = (\mathcal{S}2, 0)$*,* $(\mathcal{S}1, 1) \cup (\mathcal{S}2, 3) = (\mathcal{S}2, 3)$*.*

4.3. Implementing DFA operations

The transformation function is mainly the only operation needed to design new automaton operations. We chose the Java language to implement our package for its object-oriented programming paradigm. The abstract definition of a marked automaton `MarkedAutomaton` is designed as an interface whose contract is primarily `boolean isFinal()` and `MarkedAutomaton succ(letter a)`. As an example, the union operator is obtained by defining a corresponding new class `UnionMarkedAutomaton` implementing `MarkedAutomaton` with two at-

```
1  int nbClass;      // number of classes of the partition
2  Set [] B;         // partition of the local states
3  Set L; // L is a set of pairs (i,a) of classes and letter to be traited
4  Set [][] pred; // for all "s" local state and "a" letter : pred[s][a]={s':succ[s'][a]=s};
5
6  void refinePartitionWith (Set U) {
7      Set pivot = {i | B[i]∩U ≠ ∅∧B[i]∩U ≠B[i]};
8      sort(pivot);
9      for all "i" in pivot (in the order) {
10         B[i], B[nbClass] = B[i]\U, B[i]∩U;
11         for all "a" letter {
12             if ((i,a) ∉L and 0 < |B[i]| ≤ |B[nbClass]|)
13                 L.put(i,a);
14             else
15                 L.put(nbClass,a);
16         }
17         nbClass++;
18     }
19 }
20
21 ExitAutomaton,int [] adaptedHopcroft(ExitAutomaton exitAutomaton) {
22     nbClass = 1;
23     B[0] = local state set of exitAutomaton;
24     refinePartitionwith (final state set of exitAutomaton);
25     for all "u" exit state and all "a" letter {
26         refinePartitionwith (pred[u][a]);
27     }
28     while (!L.isEmpty()) {
29         int, int (i,a) = L.get();
30         refinePartitionWith (⋃_{s∈B[i]} pred[s][a]);
31     }
32     Assign to "minimal" to be the quotient automaton
33     Assign to "h" to be the mapping which associates to each local state its class index
34     return <minimal,h>;
35 }
```

Figure 6. Adapted Hopcroft's algorithm for exit automata

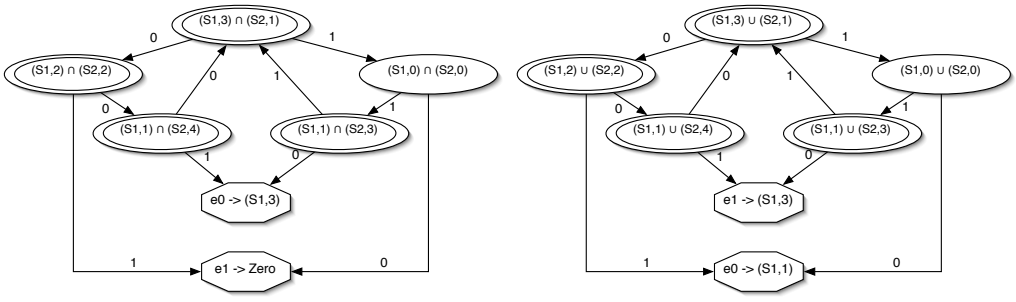
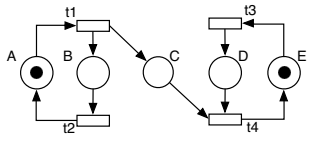


Figure 7. Examples of the evaluation of two set operations on shared automata



N	LASH	PresTaf
2	3s	2s
5	10s	4s
10	29s	10s
20	83s	27s
30	151s	48s
40	-	80s
50	-	122s

no invariant

N	LASH	PresTaf
50	16s	1s
100	38s	2s
200	89s	3s
500	256s	7s
1000	-	14s
5000	-	67s
10000	-	130s

with invariant

Table 3. Experimentation results for producer-consumer system

tributes, and by running a command like "`A=canonical(new IteMarkedAutomaton(A1,A2))`". In the same way, many new operations can be designed.

5. Experimentations

The goal is to evaluate the benefits of shared automaton techniques for verification problems. We have implemented a basic Presburger arithmetic package, named PresTaf. Binary automata have been used as a data structure to represent Presburger formula, or rather, the set of solutions to Presburger formulas. We do not describe the technique to transform a Presburger formula into an automaton. This information is not relevant to interpret the experimental results. We have mainly followed the construction proposed in [10]. It is quite natural to compare our implementation with the LASH package. Indeed, this tool has good practice for transforming Presburger formula to binary automata, while remaining traditional about automata operations. Thus, experimentation measures the benefits that our method can bring to such systems. We chose a standard verification problem for our experiments: the backward symbolic state space exploration for Petri nets. This problem is technically interesting for several reasons: the transition relation can be represented as a Presburger formula where variable states are used for place marking before or after firing a transition; backward exploration is finite for some kind of marking set (the upward closed set); the computation uses operations on Presburger formula in an intensive way. We do not claim to evaluate the most effective method, specialised studies concerning the symbolic state space exploration [2] are out of the scope of our experimentation.

Tables 3 gives backward exploration running times applied to Petri net at the left of the tables, with initial exploration marking sets $S_N = \{M : M(C) \geq N\}$. We test two types of techniques: one without any optimization, the other one where the exploration is restricted to markings fulfilling the invariant properties $M(A) + M(B) = 1$ and $M(D) + M(E) = 1$. Experimental results highlight the interest of shared automata. But applying some optimisations (as invariants) is necessary to obtain good experimental efficiency. Table 4 confirms previous experimental results when backward exploration with invariant is applied to some Petri nets. As BDD [11], the main factor of this benefits is the computed cache. Indeed, the iterations during a state space exploration share many sub-problems. It seem that shared automata are a first step towards an efficient BDD-like representation of automata for infinite state space model checking.

Model	| {Place} |	| {Transition} |	LASH	PresTaf
LEA	30	35	6min 36s	1min 13s
Manufacturing System	14	13	9min 37s	1min 4s
CSM	13	8	14min 38s	1min 2s
PNCSA	31	36	66min	3min22
ConsProd	18	14	1316min	3min55

Table 4. Experimentation results for some Petri nets

6. Conclusions

In this paper, we design the first data structure for automata manipulations based on the main BDD principles: a strong canonical form for the automata, the use of a unique table and a computed cache. Experimentions show the great benefit of the new data structure when applied in symbolic state space exploration of infinite systems. The major future work concerns the conception of new BDD-oriented structures based on arbitrary finite graphs. An interesting perspective is the application of the shared automata techniques to the symbolic structures used in MONA [6], which are based on a BDD-like representation of the transition function without any advanced sharing techniques tuned for automata.

References

[1] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST: Fast Acceleration of Symbolic Transition systems. In *CAV '03*, volume 2725 of *LNCS*, pages 118–121, 2003.

[2] C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *International Journal of Foundations of Computer Science*, 14(4):605–624, 2003.

[3] A. Bouajjani, B. Jonsson, M. Nilsson, and J. d'Orso. Algorithmic improvements in regular model checking. In *CAV '03*, volume 2725 of *LNCS*, pages 236–248, 2003.

[4] R. Bryant. Graph based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[5] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic Model Checking: 10e20 states and beyond. *Information and Computation*, 98(2):97–109, 1998.

[6] J. Elgaard, N. Klarlund, and A. Moller. Mona 1.x: new techniques for WS1S and WS2S. In *CAV '98*, volume 1427 of *LNCS*, 1998.

[7] D. Gries. Describing an algorithm by Hopcroft. *Acta Informatica*, 2:97–109, 1973.

[8] E. Hopcroft. An n log n algorithm for minimizing the states in a finite automaton. In Z. Kohavi, editor, *Theory of Machines and Computations*, pages 189–196. Academic Press, 1971.

[9] T. Knuutila. Re-describing an algorithm by Hopcroft. *Theoretical Computer Science*, 250:333–363, 2001.

[10] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *TACAS'00*, volume 1785 of *LNCS*, pages 1–19, 2000.

[11] B. Yang, R. E. Bryant, D. R. O'Hallaron, A. Biere, O. Coudert, G. Janssen, R. K. Ranjan, and F. Somenzi. A performance study of BDD-based model checking. In *FMCAD'98*, pages 255–289, 1998.