A Minimized Automaton Representation of Reachable States

Gerard J. Holzmann and Anuj Puri

Bell Laboratories
Murray Hill, NJ 07974

To appear in 'Software Tools for Technology Transfer' (Springer Verlag), 1999.

Abstract

We consider the problem of storing a set $S \subset \Sigma^k$ as a deterministic finite automaton (DFA). We show that inserting a new string $\sigma \in \Sigma^k$ or deleting a string from the set S represented as a minimized DFA can be done in expected time $O(k|\Sigma|)$, while preserving the minimality of the DFA. We then discuss an application of this work to reduce the memory requirements of a model checker based on explicit state enumeration.

Keywords: Finite Automata, Verification, OBDDs, Sharing trees, Data Compression

1 Introduction

In this paper we consider the problem of reducing the memory requirements of model checkers such as SPIN [4], that are based on an explicit state enumeration method. The memory requirements of such a model checker are dominated by state storage. Reached states are typically stored in a hash-table, mostly for efficiency reasons. We will consider here a different method for storing the states, based on the on-the-fly

construction of a deterministic finite state recognizer, that replaces the hash-table. Instead of checking for states in the hash-table, we now interrogate the recognizer. Similarly, instead of updating the hash-table, we update the recognizer if a new state is to be added to the set of reached states. The update procedure is defined in such a way that the minimality of the recognizer is always preserved. The runtime requirements of this alternative storage method can be expected to be greater than that for hash-table storage, but, as we shall see, the reduction in the memory requirements can make this a worthwhile trade-off.

A state descriptor, for the system being verified, can be thought of being encoded as a string $\sigma \in \Sigma^k$ of bits or bytes. Each string becomes a member of the reached states set S. We will discuss how one can construct and update a minimized deterministic finite automaton (DFA) that accept all the strings in set S, and none outside it.

Our main result is an algorithm for inserting and deleting strings from a DFA storing the set S. The algorithm produces a new minimized DFA which accepts exactly the new set obtained from the deletion or the insertion operation. The expected cost of the insertion and deletion operations in our algorithm is $O(k|\Sigma|)$.

We will study the effect of using this method in SPIN. Earlier, in Visser [10] an attempt was made to apply a standard BDD package [1] unmodified, to solve this problem. Similarly, Gregoire [3] experimented with a different data structure, called a sharing tree, or graph-encoded set, to achieve a similar result. We will compare our results with theirs in Section 5.

Storing a set $S \subset \Sigma^k$ as a minimized DFA is closely related to the problem of storing a boolean function as an ordered binary decision diagram (OBDD) [1, 8] (see the discussion in Section 4). The method of inserting a string σ into a set $S \subset \Sigma^k$ using a union operation has complexity $O(kM|\Sigma|)$ where M is the size of the finite automaton representing set S [1]. In contrast, the algorithm for insertion that is introduced in this paper has complexity $O(k|\Sigma|)$.

In Section 2, we define the k-layer DFA we will be using to store a set $S \subset \Sigma^k$. In Section 3, we present our algorithms for insertion and deletion. In Section 4, we discuss the relationship of our work with the previous work on OBDDs. In Section 5, we discuss the application our work to the problem of state space exploration, and compare our results to those of [3] and [10].

[1] Available from Carnegie Mellon at `emc.cs.cmu.edu` (`pub/bdd/bddlib.tar.Z`)

2 k-Layer Deterministic Finite Automata

In this section we discuss the k-layer deterministic finite automaton (DFA) which accepts a set $S \subset \Sigma^k$ of k-tuples. Each *layer* in the automaton corresponds to a bit or byte position in the state descriptor, such that the n-th layer encodes the n-th bit or byte from the state descriptor. In our implementation we will use bytes. The method itself, however, does not depend on this choice.

A k-layer DFA is $A = (\{Q_i\}_{i=0}^{k}, \delta, \Sigma)$ where Q_i is the set of states at the ith layer, $Q_0 = \{q_0\}$, $Q_k = \{0, 1\}$, Σ is the alphabet, $\delta : Q_i \times \Sigma \to Q_{i+1}$ for $0 \leq i \leq k-1$ is the transition function and $Q_i \cap Q_j = \emptyset$ for $i \neq j$. The initial state is q_0, the accepting final state is "1" and the rejecting final state is "0". Figure 1 shows a 3-layer DFA accepting the 3-tuples { 000, 001, 101 }.

Define $\sigma[i,j] = \sigma_i \sigma_{i+1} \ldots \sigma_j$. From a state $q_i \in Q_i$, a string $\sigma[i, j-1] \in \Sigma^{j-i}$ generates a run $q[i,j] = q_i q_{i+1} \ldots q_j$ where $\delta(q_m, \sigma_m) = q_{m+1}$. We define $\mathcal{L}_A(q_i) = \{ \sigma[i,k-1] \mid \sigma[i,k-1]$ generates the run $q[i,k]$ where $q_k = 1\}$. $\mathcal{L}_A(q_i)$ is the set of all strings which take q_i to the accepting final state. We define $\mathcal{L}(A) = \mathcal{L}_A(q_0)$. Similarly, we define $E_A(q_i) = \{ \sigma[0,i-1] \mid \sigma[0,i-1]$ generates the run $q'[0,i]$ such that $q'_0 = q_0$ and $q'_i = q_i \}$. $E_A(q_i)$ is the set of strings which take the state q_0 to the state q_i. When it is clear that we are talking about the DFA A, we will write $\mathcal{L}(q_i)$ and $E(q_i)$ instead of $\mathcal{L}_A(q_i)$ and $E_A(q_i)$. In the DFA of Figure 1, $\mathcal{L}(d) = \{0, 1\}$ and $E(d) = \{00\}$.

For a set $R \subset \Sigma^i$ and $\alpha \in \Sigma$, we define $\alpha \cdot R = \{\alpha \cdot \sigma \mid \sigma \in R\}$. For a set $S \subset \Sigma^k$, $\overline{S} = \Sigma^k \setminus S$. A DFA is *minimized* provided $\mathcal{L}(q_i) = \mathcal{L}(q_j)$ iff $q_i = q_j$. Equivalently, a k-layer DFA is minimized provided states which have exactly the same successors are merged together. That is, in a k-layer DFA if $\delta(q_i, \alpha) = \delta(q_j, \alpha)$ for each $\alpha \in \Sigma$ then $q_i = q_j$.

A set $S \subset \Sigma^k$ can be stored as a minimized DFA A such that $\mathcal{L}(A) = S$. Given a DFA storing a set $S \subset \Sigma^k$ and a string σ, checking whether $\sigma \in S$ is straightforward. One generates the run $q[0,k]$; $\sigma \in S$ if $q_k = 1$, otherwise $\sigma \notin S$. To obtain the minimized DFA storing $\overline{S}$, one just switches the final and non-final states in A.

From an implementation viewpoint, the DFA can be seen as a graph. Each vertex v has a field $v[\alpha]$ for each $\alpha \in \Sigma$, and a field $v[in]$. The field $v[\alpha]$ stores the vertex $\delta(v, \alpha)$ and the field $v[in]$ stores the number of incoming edges into vertex v.

Example 2.1 *Figure 1 shows a minimized DFA storing the set { 000, 001, 101 }.*

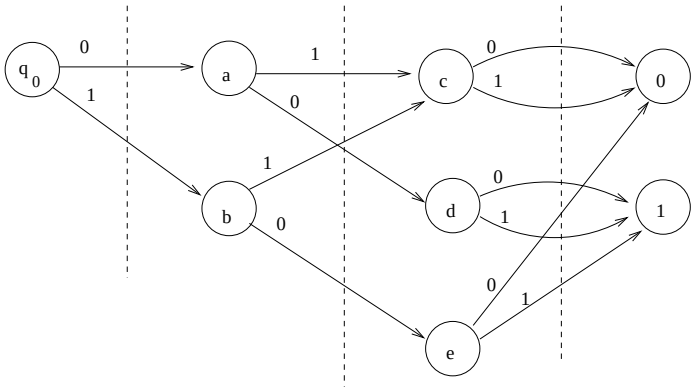

Figure 1: A minimized DFA storing the set {000, 001, 101}. Dashed lines separate states in different layers.

3 Insertion and Deletion

Given a minimized DFA A storing a set $S \subset \Sigma^k$, we may want to insert a new string $\sigma \in \Sigma^k$ into S, or delete a string σ from S. Our main result is an $O(k|\Sigma|)$ algorithm which in case of an insertion operation takes as input the automaton A and the string σ and produces a new minimized DFA A' such that $\mathcal{L}(A') = \mathcal{L}(A) \cup \{\sigma\}$; and in case of a deletion operation, it produces a new minimized DFA A'' such that $\mathcal{L}(A'') = \mathcal{L}(A) \setminus \{\sigma\}$.

3.1 Insertion

Given the k-layer minimized DFA A and a string $\sigma \in \Sigma^k$, the insertion procedure produces a new minimized DFA A' such that $\mathcal{L}(A') = \mathcal{L}(A) \cup \{\sigma\}$.

For a string σ, we generate the run $q[0,k]$. The insertion procedure is based on the following observation: if $\sigma \notin \mathcal{L}(A)$ then in the new DFA A', we need a state n_i in each layer i such that $\mathcal{L}_{A'}(n_i) = \mathcal{L}_A(q_i) \cup \{\sigma[i,k-1]\}$ and $\sigma[0,i-1] \in E_{A'}(n_i)$.

The DFA A' is constructed from the DFA A by adding some new states and deleting others. If there is already a state n_i in layer i of A so that $\mathcal{L}_A(n_i) = \mathcal{L}_A(q_i) \cup \{\sigma[i,k-1]\}$, then in the new DFA A', $\mathcal{L}_{A'}(n_i) = \mathcal{L}_A(q_i) \cup \{\sigma[i,k-1]\}$, $E_{A'}(n_i) = E_A(n_i) \cup \{\sigma[0,i-1]\}$ and $E_{A'}(q_i) = E_A(q_i) \setminus \{\sigma[0,i-1]\}$. The state q_i is deleted in A' if $E_{A'}(q_i) = \emptyset$.

If there is no state n_i in layer i of DFA A such that $\mathcal{L}_A(n_i) = \mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$, then we need to create such a state. If $E_A(q_i)$ contains only $\sigma[0, i-1]$, then in the new DFA A' we modify things so that $\mathcal{L}_{A'}(q_i) = \mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$. But if $E_A(q_i)$ contains more than one element, then in the DFA A' we create a new state q_i' such that $\mathcal{L}_{A'}(q_i') = \mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$, $E_{A'}(q_i') = \{\sigma[0, i-1]\}$ and $E_{A'}(q_i) = E_A(q_i) \setminus \{\sigma[0, i-1]\}$.

Example 3.1 *Consider the DFA A of Figure 1 that is storing { 000, 001, 101 }. Suppose we want to insert $\sigma = 100$ into A. The run of σ in A is $q[0,3] = (q_0, b, e, 0)$. Now $\mathcal{L}_A(n_2) = \mathcal{L}_A(e) \cup \{0\}$ for $n_2 = d$ and $\mathcal{L}_A(n_1) = \mathcal{L}_A(b) \cup \{00\}$ for $n_1 = a$. Since $E_A(b) = \{1\}$ and $E_A(e) = \{10\}$ are singletons, these states will be deleted in the new automaton. Figure 2 shows the new DFA A'. Notice that in the new DFA A', $E_{A'}(a) = E_A(a) \cup \{1\} = \{0, 1\}$ and $E_{A'}(d) = E_A(d) \cup \{10\} = \{00, 10\}$.*

We now describe the algorithm for the insertion operation in detail. The insertion operation is divided into three phases. The first phase generates the run $q[0, k]$ of σ in A. If $q_k = 1$ then $\sigma \in \mathcal{L}(A)$ and we are done; otherwise $\sigma \notin \mathcal{L}(A)$ and we go to Phase 2.

Phase 1

Generate run $q[0, k]$ of σ
If $q_k = 1$
then Stop
else Goto Phase 2

As mentioned previously, at each layer i in the new DFA A', we must have a state n_i such that $\mathcal{L}_{A'}(n_i) = \mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$ (because after applying $\sigma[0, i-1]$ in A', we must reach a state which accepts exactly $\mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$). The purpose of Phase 2 is to check if such states are already present in our original DFA A. In particular, Phase 2 finds a state v in layer $m+1$ of DFA A such that $\mathcal{L}_A(v) = \mathcal{L}_A(q_{m+1}) \cup \{\sigma[m+1, k-1]\}$, and there is no state at layer j of DFA A with this property for $j \leq m$.

This is done iteratively. If the state *old* in layer $i+1$ has the property that $\mathcal{L}_A(old) = \mathcal{L}_A(q_{i+1}) \cup \{\sigma[i+1, k-1]\}$, then we try to find a state *new* in layer i such that $\mathcal{L}_A(new) = \mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$ (because A is minimized, there can be at most one state in layer i with this property). Given *old* and q_i, the key to finding the state *new* is to observe that $\delta(new, \sigma_i) = old$ and $\delta(new, \alpha) = \delta(q_i, \alpha)$ for $\alpha \in \Sigma \setminus \{\sigma_i\}$.

Definition 3.1 *Given the set* $K = \{\langle q_\alpha, \alpha\rangle | \alpha \in \Sigma\}$, *we define* $pred(K) = \{q |\ \forall \alpha\, \delta(q,\alpha) = q_\alpha\}$.

Notice that if A is a minimized DFA then $new = pred(K)$ contains at most one element. And if $new \neq \emptyset$, then $\delta(new, \alpha) = q_\alpha$ for each $\alpha \in \Sigma$.

Phase 2

i := k
new := {1}
Repeat
    i := i - 1
    old := new
    new := pred $(\{\langle q_i[\alpha], \alpha\rangle | \alpha \in \Sigma \setminus \{\sigma_i\}\} \cup \{\langle old, \sigma_i\rangle\})$
Until (new = $\emptyset$*)*
m := i
v := old
Goto Phase 3

Lemma 3.1 *At the end of the "Repeat-Until" loop of Phase 2, the variables new and i satisfy the following loop invariant: if* $new \neq \emptyset$ *then* $|new| \leq 1$ *and* $\mathcal{L}_A(new) = \mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$; *and if* $new = \emptyset$ *then for each* $r \in Q_i$, $\mathcal{L}_A(r) \neq \mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$.

Proof: Suppose $\mathcal{L}_A(new) = \mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$ at the beginning of the loop. Then at the end of the loop $\mathcal{L}_A(old) = \mathcal{L}_A(q_{i+1}) \cup \{\sigma[i+1, k-1]\}$. If $new \neq \emptyset$ at the end of loop, then from the previous discussion $|new| \leq 1$, $\delta(new, \sigma_i) = old$ and $\delta(new, \alpha) = \delta(q_i, \alpha)$ for $\alpha \in \Sigma \setminus \{\sigma_i\}$. Hence

$$\begin{aligned}
\mathcal{L}_A(new) &= \cup_{\alpha \in \Sigma \setminus \{\sigma_i\}} \{\alpha \cdot \mathcal{L}(\delta(q_i, \alpha))\} \cup \{\sigma_i \cdot \mathcal{L}(old)\} \\
&= \cup_{\alpha \in \Sigma \setminus \{\sigma_i\}} \{\alpha \cdot \mathcal{L}(\delta(q_i, \alpha))\} \cup \{\sigma[i, k-1]\} \cup \{\sigma_i \cdot \mathcal{L}(q_{i+1})\} \\
&= \cup_{\alpha \in \Sigma} \{\alpha \cdot \mathcal{L}(\delta(q_i, \alpha))\} \cup \{\sigma[i, k-1]\} \\
&= \mathcal{L}(q_i) \cup \{\sigma[i, k-1]\}
\end{aligned}$$

But if $new = \emptyset$ then there is no state $r \in Q_i$ such that $\delta(r, \sigma_i) = old$ and $\delta(r, \alpha) = \delta(q_i, \alpha)$ for $\alpha \in \Sigma \setminus \{\sigma_i\}$. Hence for every state $r \in Q_i$, $\mathcal{L}_A(r) \neq \mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$. ■

Phase 3 constructs the new minimized DFA A' such that $\mathcal{L}(A') = \mathcal{L}(A) \cup \{\sigma\}$. At each layer i of A' there will be a state n_i such that $\mathcal{L}_{A'}(n_i) = \mathcal{L}_A(q_i) \cup \{\sigma[i, k-1]\}$. We will construct the new DFA A' from the DFA A by adding some new states and deleting others.

For layers $i \geq m+1$ we already have states in A with this required property, so we only need to create such states for $j \leq m$. We do this by looking at the state q_j at layer j. If q_j has more than one incoming edge (i.e., $E_A(q_j)$ has more than one element), then in the new DFA A', we make a duplicate of q_j called q'_j. Since q_j has more than one incoming edge, $\mathcal{L}_A(q_j)$ is needed in layer j of the DFA A'. So it will be the new state q'_j which will have the property that $\mathcal{L}_{A'}(q'_j) = \mathcal{L}_A(q_j) \cup \{\sigma[j, k-1]\}$. But if q_j has at most one incoming edge (i.e., $E_A(q_j)$ contains only $\sigma[0, i-1]$), then $\mathcal{L}_A(q_j)$ is no longer needed in the DFA A'. So it will be the state q_j in A' which will have the property that $\mathcal{L}_{A'}(q_j) = \mathcal{L}_A(q_j) \cup \{\sigma[j, k-1]\}$.

The function $Remove()$ deletes the state q_i for $i \geq m+1$ provided $E_A(q_i)$ contains only $\sigma[0, i-1]$.

Phase 3

$s := q_0$
for $j = 1$ *to* m {
    if $q_j[in] > 1$ *then*
        Create vertex q'_j
        $q'_j[\alpha] := q_j[\alpha]$ *for each* $\alpha \in \Sigma$
        for each $\alpha \in \Sigma$
            $(q'_j[\alpha])[in] := (q'_j[\alpha])[in] + 1$
        $q_j[in] := q_j[in] - 1$
        $s[\sigma_{j-1}] := q'_j$
        $q'_j[in] := 1$
        $s := q'_j$
    else
        $s := q_j$
}
$rem := s[\sigma_m]$
$s[\sigma_m] := v$
$v[in] := v[in] + 1$
$rem[in] := rem[in] - 1$
$Remove(rem)$

Remove (rem):

Starting from rem, delete all vertices which have no incoming edges

Lemma 3.2 *The new DFA A' is minimized and $\mathcal{L}(A') = \mathcal{L}(A) \cup \{\sigma\}$.*

Proof: Suppose s_j is the value of variable s at the end of the jth iteration of the "for-loop" in Phase 3. Then it can be checked that if $\mathcal{L}_{A'}(s_j) = \mathcal{L}_A(q_j) \cup \{\sigma[j, k-1]\}$ then $\mathcal{L}_{A'}(s_{j-1}) = \mathcal{L}_A(q_{j-1}) \cup \{\sigma[j-1, k-1]\}$. Because $\mathcal{L}_{A'}(s_m) = \mathcal{L}_A(q_m) \cup \{\sigma[m, k-1]\}$, it follows that for $j \leq m$, $\mathcal{L}_{A'}(s_j) = \mathcal{L}_A(q_j) \cup \{\sigma[j, k-1]\}$. In particular, $\mathcal{L}_{A'}(q_0) = \mathcal{L}(A') = \mathcal{L}(A) \cup \{\sigma\}$.

To show that the new DFA A' is already minimized, we need to observe that each state except q_0 has an incoming edge, and for two different states r and s in A', $\mathcal{L}_{A'}(r) \neq \mathcal{L}_{A'}(s)$. ∎

Notice that the DFA A' has at most k more states than the DFA A.

Example 3.2 *Let us insert the string 100 into the set {000,001,101} represented by the DFA of Figure 1. We will go through the Phases 1-3 to construct the new DFA A'. In Phase 1, we generate the run $q[0,3] = (q_0, b, e, 0)$. Since $q_3 = 0$, we go to Phase 2. At the beginning of the first iteration of the "Repeat-Until" loop, $i = 3$ and $new = \{1\}$. In the next iteration, we get $i = 2$ and $new = pred(\langle q_2[1], 1\rangle\langle 1, 0\rangle) = pred(\langle 1, 1\rangle\langle 1, 0\rangle) = \{d\}$. In the next iteration, $i = 1$ and $new = pred(\langle q_1[1], 1\rangle\langle d, 0\rangle) = pred(\langle c, 1\rangle\langle d, 0\rangle) = \{a\}$. During the next iteration, $i = 0$ and $new = pred(\langle q_0[0], 0\rangle\langle a, 1\rangle) = pred(\langle a, 0\rangle\langle a, 1\rangle) = \emptyset$. The "Repeat-Until" loop terminates, the value of $m = 0$ and $v = a$. Notice $\mathcal{L}(a) = \mathcal{L}(b) \cup \{00\}$. In Phase 3, $s = q_0$. The "for-loop" is not executed because $m = 0$. The value of $rem = q_0[1] = b$. The new value of $q_0[1] = a$. And we call the procedure Remove($\{b\}$) which deletes the vertex b, and then the vertex e. The resulting DFA is shown in Figure 2. Notice that it is already minimized and it accepts exactly the set {000,001,100,101}.*

3.2 Deletion

Given a k-layer minimized DFA A and a string $\sigma \in \Sigma^k$, the deletion operation produces a minimized DFA A'' such that $\mathcal{L}(A'') = \mathcal{L}(A') \setminus \{\sigma\}$.

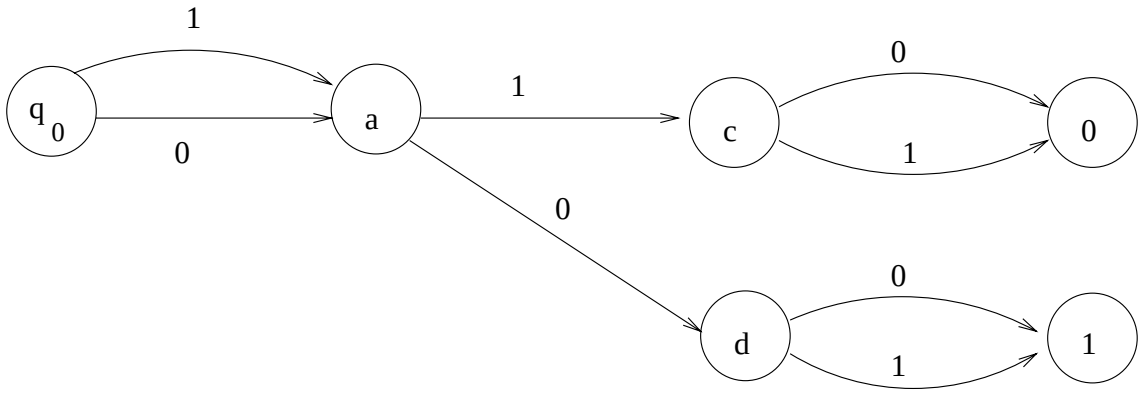

Figure 2: The new minimized DFA storing {000,001,100,101}

To perform the deletion operation, we first complement A to obtain the DFA C. We then insert σ into C obtaining the DFA C' so that $\mathcal{L}(C') = \mathcal{L}(C) \cup \{\sigma\}$. We obtain the DFA A'' by complementing C'. So $\mathcal{L}(A'') = \overline{(\overline{\mathcal{L}(A)} \cup \{\sigma\})} = \mathcal{L}(A) \setminus \{\sigma\}$.

Deletion

C = Complement of A
Construct C' using the insertion operation
so that $\mathcal{L}(C') = \mathcal{L}(C) \cup \{\sigma\}$
A'' = Complement of C'

Example 3.3 *Let us delete the string 100 from the set $\{000, 001, 100, 101\}$ represented by the DFA of Figure 2. We first complement this DFA to obtain the DFA C shown in Figure 3.*

We next insert the string 100 into the DFA C by going through Phases 1-3 of the insertion procedure. In Phase 1, we generate the run $q[0,3] = (q_0, a, d, 0)$. Since $q_3 = 0$, we go to Phase 2. At the beginning of the first iteration of the "Repeat-Until" loop, $i = 3$ and $new = \{1\}$. In the next iteration, $i = 2$ and $new = pred(\langle q_2[1], 1\rangle, \langle 1, 0\rangle) = pred(\langle 0, 1\rangle\langle 1, 0\rangle) = \emptyset$. The "Repeat-Until" loop terminates, the value of $m = 2$ and $v = 1$. In Phase 3, in the beginning of the first iteration of the "for-loop" $s = q_0$. Since $q_1 = a$ has more than one incoming edge, we make a duplicate state a' and make $q_0[1] = a'$. In the next iteration, $s = a'$, and since $q_2 = d$ now also has more

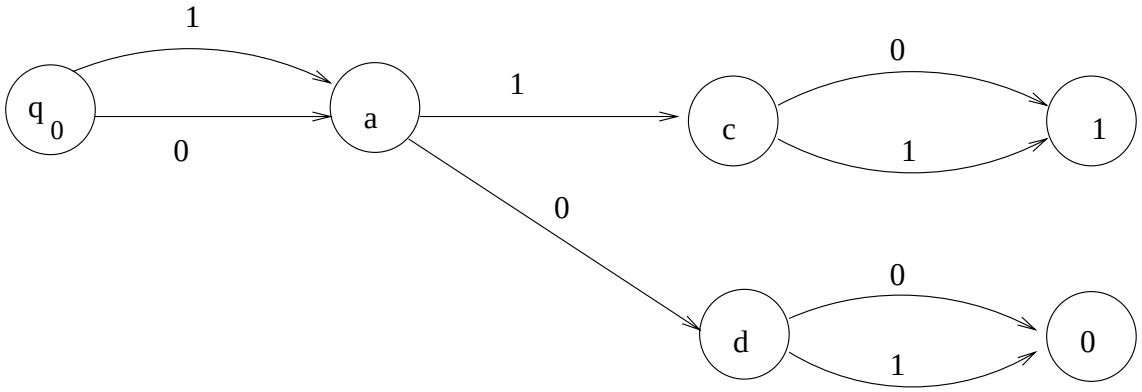

Figure 3: The DFA C

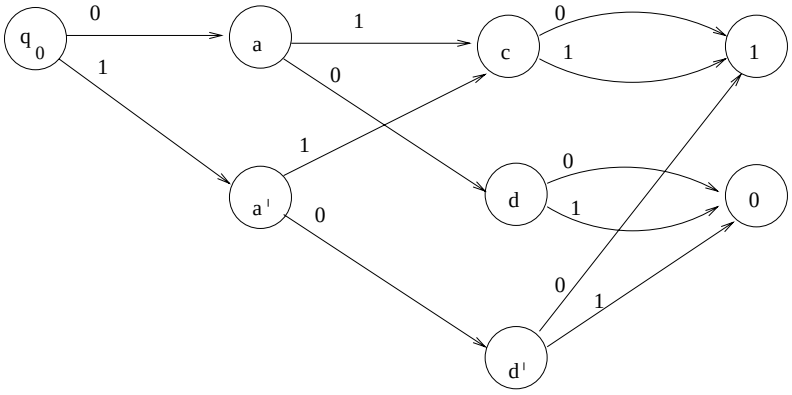

Figure 4: The DFA C'

than one incoming edge, we make a duplicate state d' and make $a'[0] = d'$. The new value of $s = d'$. The "for-loop" terminates and the new value of $d'[0] = 1$. The gives us the DFA C' shown in Figure 4.

To obtain the DFA A'', we complement the DFA C'.

3.3 Complexity

We next analyze the cost of the insertion procedure. Phase 1 takes $O(k)$ time. In Phase 2, in each iteration of the "Repeat-Until" loop, we need to compute $new = pred(K)$ for a set K. This is done using a hash table. For each state q, we store q in the hash table using the key $K = \{(q[\alpha], \alpha) | \alpha \in \Sigma\}$. In computing $new = pred(K)$, K is used as the key. Assuming constant-time hashing, $pred(K)$ takes $O(|\Sigma|)$ expected time. So Phase 2 takes $O(k|\Sigma|)$ expected time. Phase 3 takes $O(k|\Sigma|)$ time. So the expected cost of an insertion operation is $O(k|\Sigma|)$ time. The cost of a deletion operation is the same as the cost of an insertion operation.

4 Relationship with OBDDs

There is a close relationship between a minimized k-layer DFA storing a set $S \subset \Sigma^k$ and an OBDD representing a boolean function. A boolean function $f : \{0,1\}^k \rightarrow \{0,1\}$ is characterized its truth set $\chi = \{\sigma \in \{0,1\}^k | f(\sigma) = 1\}$. The truth set χ can then be stored as a minimized k-layer DFA where the alphabet is $\Sigma = \{0,1\}$. This is a canonical representation of the boolean function f. This method of representing a boolean function is essentially equivalent to representing the boolean function as an OBDD (see [8]).

In a minimized DFA, a state s is "redundant" when there is a state t such that $t = \delta(s,0) = \delta(s,1)$. Such redundant states can be removed by redirecting incoming edges into s to t. An OBDD for a boolean function is obtained from a minimzed DFA that is storing its truth set by removing all redundant states.

Example 4.1 *Figure 5 shows the OBDD of the boolean function with truth set {000, 001, 101}. It is obtained from the minimized DFA of Figure 1 by removing the redundant states c and d.*

Given the OBDDs for boolean functions f and g, the OBDD for $f + g$ can be

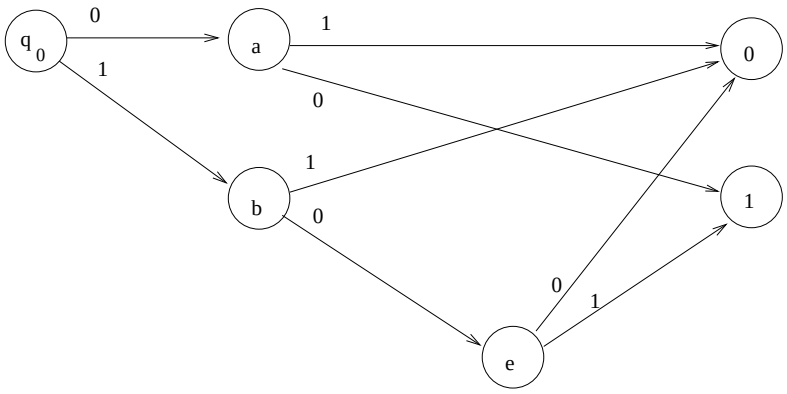

Figure 5: An OBDD with truth set {000, 001, 101}

computed in time $O(mn)$ where m is the size of the OBDD for f and n is the size of the OBDD for g [1, 2]. Given minimized k-layer DFAs A and B, the same construction can be used to compute the minimized DFA C such that $\mathcal{L}(C) = \mathcal{L}(A) \cup \mathcal{L}(B)$.

Theorem 4.1 *[1, 2] Given k-layer minimized DFAs A and B, the minimized DFA C such that $\mathcal{L}(C) = \mathcal{L}(A) \cup \mathcal{L}(B)$ can be computed in expected time $O(mn|\Sigma|)$ where m is the size of the DFA A, n is the size of the DFA B and Σ is the alphabet.*

It also follows that given the DFAs A and B of size m and n, the minimized DFA C such that $\mathcal{L}(C) = \mathcal{L}(A) \cap \mathcal{L}(B)$ can be computed in expected time $O(mn|\Sigma|)$.

5 Application

The SPIN verification system uses an on-the-fly LTL model checking procedure based on explicit state enumeration [6]. Reachable system states are compressed [5], and stored in a hash-table as simple byte-sequences. The maximum amount of memory available to the model checker sets a trivial upper bound to the maximum state space size that can be explored in this way.

The search strategy itself is a modified depth-first search, described in more detail in [7]. To perform the model checking efficiently, we need to be able to recognize quickly which reachable system states were explored before in the search, and which of these states are currently on the depth-first search stack.

	Method	States	Mem.(Mb)	Time(sec.)
0	Reference (no compression)	251409	37.401	60.91
1	Direct DFA implementation	251409	25.113	384.33
2	Intermediate version	251409	15.994	3606.14
3	Final Implementation	251409	6.778	107.08

Table 1: Effect of implementation choices, a data transfer protocol model.

We first consider some of the implementation choices we have made to optimize the performance of the algorithm, when used within a model checking system. Next we compare the performance of the algorithm we have described with several others that have the same objective.

5.1 Implementation in SPIN

The algorithm described in the previous sections is used to provide the two central state maintenance functions in SPIN, optionally replacing standard hash-table lookups of states compressed by more conventional means. The first function is to identify whether a newly generated system state was previously visited in the search. The second function is to identify whether a previously visited state is currently on the depth-first search stack, implying that not all of its successors have yet been explored.

The performance of the DFA encoding algorithm is sensitive to the particulars of the implementation. To illustrate this, consider the measurements presented in Table 1. All measurements reported here are for a 180 MHz SGI Workstation with 64 Megabytes of main memory. Only the relative performance of the various algorithms, however, will be of interest to us here.

Row 0 in Table 1 gives, for reference, the performance of an exhaustive search without compression for a reference protocol model that generates 251,409 reachable states. Memory use, for all measurements, is reported in Megabytes, time is reported in seconds of CPU time.

Row 1 gives the memory and time requirements for a direct implementation of the algorithm given in the preceding section, coded in about 300 lines of C.

Row 2 shows the results of a first attempt to reduce the memory requirements by changing from a bit-level encoding to a byte-level encoding per layer (more about this below).

Row 3 shows the performance of the final version of the algorithm, with several other optimizations in place.

The optimizations made in the final implementation can be summarized as follows:

1. Byte level encoding,
2. Range compression on edges,
3. Splay trees with linked lists for backward pointers,
4. Byte tags on the trailing symbol in each word, for special state markings.
5. Edge lists organized with the most recently accessed edge appearing first.

Some of these methods have been tried before in different contexts. Specifically, splay trees and byte tags were also used in [3]. We briefly comment on each point below.

Byte Level Encoding and Range Compression

The model checker SPIN targets software verification problems, that is the verification of the interactions of multiple, asynchronously executing, processes that communicate either via message buffers or shared variables [6]. The data objects in this type of application can be defined at the bit-level, but they are more commonly defined at a higher level of abstraction, e.g., as integer variables, or as user defined data structures. This means that a bit-level encoding, as commonly used in hardware applications, is not necessarily optimal in this application.

The implementation we have built for the SPIN system therefore uses a byte-level encoding of the algorithm. Every symbol in the word that is stored in the DFA represents a byte of data from the state descriptor. This implies that every node in the DFA has not two but up to 256 possible outgoing edges. To capitalize on the fact that most of the edges of a node typically lead to the same set of successor nodes, we have implemented *range* encoding for the edges in the graph. Every consecutive

range of edges with a common destination can thus be stored in a single edge data structure, thus reducing the overall memory requirements.

Splay Trees

In the second phase of the algorithm, the graph structure has to be searched backwards, from the accepting node towards the root node. As with BDDs, the number of backward pointers per node can be substantially larger than the number of forward pointers per node. Therefore, we have adopted an encoding method that is similar to the one that is often used for BDDs. Instead of hash-table lookup, however, we have used a variation of splay trees, based on an algorithm from [9].

The reason for using splay tree code is to exploit predictable access patterns to the graph structure, to improve performance. We compute the keys for the splay tree storage from the addresses of the predecessor nodes in the DFA. In the worst case, there can be 256 such addresses to consider in the computation of these keys. By caching some of the information inside the nodes, however, this can be reduced to fairly simple constant time computation. The splay tree code from [9] assumes that no two items with equal keys can appear within the same tree. This is not true for the keys that we use, so we adapted the code by adding an optional linked list on each splay tree node, where equal size keys can be stored and found. The resulting modification of the splay tree algorithm is minimal, and since equal keys are sufficiently seldom, the effect on the runtime is also minimal.

In the implementation of the graph encoding method for state vector storage described in [3] a different implementation of splay trees was used to store forward instead of backward edges in the graph-encoded sets. Equal keys can appear also here, and in this case the conflicts are resolved without linked lists within the existing tree structure. This calls for a greater modification of the splay tree algorithm, but also no significant effect on runtime.

Byte Tags

To differentiate between states that are on the depth-first search stack, and those that have been removed from it, we use the method given in [3]. Each state carries a tag as the last byte. The first time the state is encountered, the tag is zero, and the state is encoded as such in the DFA. When a state is removed from the search stack, the tag becomes non-zero, and the state is stored again, with maximal sharing of information between the two copies of the state. Checking with the member function

	Algorithm	States	Nodes	Mem. (Mb)	Time (sec.)
1	No Compression [6]	417321	–	63.2	18.61
2	Collapse [5]	417321	–	11.7	43.56
3	DFA	417321	156744	7.6	201.05
4	DFA + Collapse	417321	17486	3.5	200.49
5	GETSs [3]	417321	166833	7.4	225.03
6	OBDDs + Collapse [10]	417321	357247	13.7	3463.72

Table 2: Measurements and Comparison for a file transfer protocol.

for the presence of the second copy of the state in the DFA can be done efficiently, since the previously constructed path through the DFA can be reused, upto, but not including, the last symbol. We use the same technique to store also a *proviso* bit, required for partial order searches as defined in [7], without noticeable overhead.

Dynamic Reordering of Edge Lists

The most recently created part of the DFA has the highest probability of being revisited in subsequent accesses of the structure. Performance is optimized under these circumstances by dynamically reorganizing the edge lists on each node with the most recently created or accessed edge appearing first.

5.2 Comparisons

Table 2 shows the effect on run-time and memory use of replacing the standard hash-table for explicit state storage in SPIN with the DFA based encoder discussed in this paper for a model of a sliding window flow control protocol, taken from [4]. The first row gives memory use and run-time when storing all reachable states in a standard hash-table, without compression. The second row is for a run with SPIN's builtin COLLAPSE compression algorithm, which is discussed in more detail in [5]. The third row gives the results when the DFA encoding from this paper is used, and the fourth row shows the results when the state descriptors are first compressed with COLLAPSE, and then stored with the DFA method from this paper.

A few trends are visible. First, as may be expected, applying a compression tech-

nique will cause an increases in the run-time requirements. Greater compression in these cases is paired with greater run-time overhead. A direct application of the DFA encoding reduces the memory requirements from 63.2 to 7.6 Mb, or by approximately 88 percent. The runtime, however, increases tenfold. Combining the COLLAPSE compression mode with DFA encodings reduces the memory requirements further to 3.5 Mb, giving a reduction of approximately 95 percent, without affecting the runtime requirements much. The reason for the latter is that the pre-compression of the state descriptor reduces its length from 118 bytes to 11 bytes, which speeds up the DFA encoder sufficiently to make up for the pre-compression overhead.

Similar studies with graph or BDD-based encoding techniques were reported in [3] and [10]. It is difficult to make direct comparisons with the measurements that were published in this earlier work. The numbers for memory use reported in Table 2 are totals for all memory use consumed in the verifier, both for the state encoding itself and for extraneous data structures. In [3] and [10] some of these quantities were excluded from the data reported, and in different ways in the two papers. In the measurements reported in [10], memory use was estimated by multiplying the number of nodes in the BDDs with the average number of bytes consumed per node, and no measurement was made of physically allocated memory. The memory use in [3] was also estimated instead of measured on calls to the memory allocator.

Both Visser and Gregoire, however, have generously made their C-code implementations available, so that a direct comparison could be done on a single workstation (a 180 MHz SGI system with 64 Mbyte of main memory), reporting runtimes and overall memory consumptions in the same way for each algorithm, when compiled and run with matching parameters. Table 2 gives the counts of actual physical memory allocated for each algorithm.

Each algorithm was run in exhaustive search mode, without partial order reduction, with the minimal settings for stack-depth, hash-table size, graph-depth, or BDD-size, as appropriate.

Row 5 in Table 2 reports the results of a direct application of the graph encoded sets as described in [3], without other compression methods applied. The result should be compared with the direct application of the DFA encoding, from row 3.

Row 6 reports the results of the application of vintage BDDs with COLLAPSE pre-compression, as described in [10]. The result should be compared with row 4.

A small, but mostly insignificant, difference in run-times, and a similarly small difference in the memory requirements can be noted between the GETS method (row

Algorithm	Average Mem. (Mb.)	Average Time (sec.)
Uncompressed	24.04	12.61
Collapse	8.48	16.94
DFA+Collapse	4.35	78.17
DFA	3.37	61.82
GETS	3.30	65.44

Table 3: Memory and Time *Averaged* over Fourteen Typical Applications

5) and the DFA method (row 3). There is a more significant difference between the method based on OBDDs (row 6) and the DFA encoding (row 4), combined with COLLAPSE compression, both in run-time and in memory use.

Clearly, all methods achieve the goal of memory reduction. Within the context of the SPIN model checker, the classic BDD package, with bit-level instead of byte-level, encodings, appears not to be competitive. The difference in run-times is approximately a factor of 17, and the difference in memory requirements approximately a factor of four. The graph encoded sets and the DFA encoding method produce closely competitive results.

Experiments with a range of other models confirm the trends observed above, with minor variations. Table 3 shows the results of averaging the memory use and time requirements of fourteen runs for as many different randomly chosen models, from a database of SPIN applications. The lowest average memory use is reported for the GETS code. The DFA algorithm described in this paper comes within 2 percent of that result, and lowers the runtime by about 5 percent. The COLLAPSE method gives a respectable compression for a modest runtime penalty. On average, the combination of DFA with COLLAPSE does not improve the reduction.

Table 4 compares the performance of the DFA algorithm with or without COLLAPSE pre-compression, and COLLAPSE compression applied separately directly for the same fourteen applications. The best memory reduction achieved is shown in the second column, a the linear factor of memory use divided by compressed memory use. The corresponding time penalty is given in the third column. The last column notes which algorithm came out best for each of the fourteen applications.

	Algorithm	Memory Reduction	Time Increase
1	DFA	5.15	3.99
2	DFA	5.88	11.99
3	DFA	7.72	9.31
4	DFA	9.03	9.75
5	DFA	9.50	9.29
6	DFA	20.03	4.97
7	DFA	35.09	5.79
8	DFA	66.5	8.15
9	DFA	128.50	6.77
10	DFA+Collapse	2.17	8.12
11	DFA+Collapse	3.25	8.98
12	DFA+Collapse	4.62	7.18
13	DFA+Collapse	16.43	8.44
14	DFA+Collapse	17.96	5.57

Table 4: Best Memory Reduction Factor Measured for Each of Fourteen Applications with the Corresponding Time Increase Factors

In most cases, the DFA encoding gave the best memory reduction when applied alone. In five cases the combination with COLLAPSE pre-compression improved the reduction.

In six cases, the memory reduction was a factor of ten or more, and it was never less than a factor of two. The worst time penalty was a near a factor of ten, and never less than a factor of about four.

5.3 An Extension: Checkpointing

Large SPIN verification runs typically complete within minutes of CPU time, or in the worst cases within a few hours. An increase of the run time requirements by an order of magnitude, however, can turn minutes into hours, and hours into days. Under those circumstances, it can be important to have a checkpointing option, where a snapshot of the verification results can be written to disk once in a while. In case a very long run has to be aborted, the run can then be restarted from the last checkpoint file that was written.

With explicit state storage methods, and the speed of in-core on-the-fly methods, checkpointing algorithms are self-defeating: it takes more time to read the state descriptors from a disk-file than it takes to recompute them.

The DFA storage method changes this behavior of the verifier. Not only does it make checkpointing more desirable, it also makes it more feasible. Especially for the higher compression ratios, near two orders of magnitude, writing or reading a disk-file representation of the graph can now be done in less time than it takes to recompute the graph. We have therefore extended the DFA algorithm with a checkpointing option in SPIN.

If the option is selected, a checkpointing file is written during the verification process at every multiple of one million states stored in the DFA. The nodes are identified in the checkpoint file by their memory address (a convenient wordsize integer that uniquely identifies each node). The nodes are written layer by layer, and each node is followed by a list of its successors, identified again by their address, at the next layer in the graph.

To reconstruct the graph, when restarting from a checkpoint file, is relatively straightforward. There is just one unusual problem to be solved: the graph contains both the states that are off the depth-first search stack at the moment that the checkpoint file was written, and states that are still on it. The stack itself cannot

be reconstructed from this information (the depth at which is a state is encountered is normally not available), therefore, before we can restart the verification run, the stack states have to be removed from the graph.

The removal of the stack states is done in three steps. Assume that the byte tag that identifies off-stack states has the non-zero value OFF.

Step 1

Find the node r at layer $k-1$ with
$\delta(r,0) = 1$ and $\delta(r, OFF) = 0$
If there is no such node
then Stop
else Goto Step 2

All paths in the graph between q_0 and r, followed by 0, correspond to the words that should be deleted from the graph. Note that there can be at most one node r with the desired properties.

Step 2

Make r the root of a tree with successor function δ' defined as follows:
i := k - 1
Repeat
i := i - 1
$\delta'(r, \sigma) := r'$
for each $\sigma \in \Sigma$ with $r' \in \{q | \delta(r', \sigma) = r\}$
r := r'
Until (i = 0)

After Step 2, the tree rooted in r contains all words (state descriptors) corresponding to states that were on the depth-first search stack at the time the checkpoint file was written. The tree is $k-1$ layers deep, and every path from a leaf of the tree to the root at r is such a word, when suffixed by a zero tag.

Step 3

Generate the paths from each leaf to the root of the new tree,
e.g., with a depth first search of the tree.
Append a zero to each path.
Delete the corresponding words from the graph.

Note that the information in the tree cannot be corrupted by the modifications of the graph while the words are being deleted. In a final step, the memory used for the tree structure can be reclaimed and the verification can be restarted with the pruned graph as the initial state space.

6 Conclusion

We have considered a method for representing finite words in the form of a deterministic, minimized, automaton, and have given an algorithm for the insertion of new words into the automaton, that preserves the automaton's minimality.

The application of this storage technique in a model checking tool causes an increase of the runtime requirements by about an order of magnitude, but in most cases it succeeds in reducing the memory requirements by the same amount.

Acknowledgements:

The authors are grateful to W. Visser and J-C. Gregoire for making their software available for the measurements reported in this paper, and to Mihalis Yannakakis for insightful comments on an earlier draft of this paper.

References

[1] R.E. Bryant. *Graph based algorithms for boolean function manipulation*, IEEE Trans. on Computers, C-35, pp. 677-691, (1986).

[2] R.E. Bryant. *Symbolic boolean manipulation with ordered binary decision diagrams*, Computing Surveys Volume 24, No. 3, September 1992, pp. 293-318.

[3] J-C. Gregoire. *State space compression in* SPIN *with GETSs*, Proc. Second Spin Workshop, Rutgers University, New Brunswick, New Jersey, August 1996, American Mathematical Society, DIMACS/32.

[4] G.J. Holzmann. *Design and Validation of Computer Protocols*, Prentice Hall Software Series, 1991.

[5] G.J. Holzmann. *State compression in* SPIN, Proc. Third Spin Workshop, Twente University, The Netherlands, April 1997.

[6] G.J. Holzmann. *The model checker* SPIN, IEEE Trans. on Software Engineering, Vol. 23, No. 5, May 1997.

[7] G.J. Holzmann, D. Peled, and M. Yannakakis. *On nested depth first search*, Proc. Second Spin Workshop, Rutgers University, New Brunswick, New Jersey, August 1996, American Mathematical Society, DIMACS/32.

[8] S. Kimura, and E.M. Clarke. *A parallel algorithm for constructing binary decision diagrams*, In 1990 IEEE Int. Conf. on Computer Design, Sept. 1990.

[9] D. Sleator, and R. Tarjan. *Self-adjusting Binary Search Trees*, JACM Volume 32, No 3, July 1985, pp 652-686.

[10] W. Visser. *Memory efficient storage in* SPIN, Proc. Second Spin Workshop, Rutgers University, New Brunswick, New Jersey, August 1996, American Mathematical Society, DIMACS/32.