Implementing Reversed Alternating Finite Automaton (r-AFA) Operations*

Sandra Huerter, Kai Salomaa, Xiuming Wu, and Sheng Yu

Department of Computer Science
University of Western Ontario
London, Ontario, Canada N6A 5B7
{huerter,ksalomaa,wu6,syu}@csd.uwo.ca

Abstract. In [17], we introduced a bit-wise representation of r-AFA, which greatly improved the space efficiency in representing regular languages. We also described our algorithms and implementation methods for the union, intersection, and complementation of r-AFA. However, our direct algorithms for the star, concatenation, and reversal operations of r-AFA would cause an exponential expansion in the size of resulting r-AFA for even the average cases. In this paper, we will design new algorithms for the star, concatenation, and reversal operations of r-AFA based on the bit-wise representation introduced in [17]. Experiments show that the new algorithms can significantly reduce the state size of the resulting r-AFA. We also show how we have improved the DFA-to-AFA transformation algorithm which was described in [17]. The average run time of this transformation using the modified algorithm has improved significantly (by 97 percent).

1 Introduction

The study of finite automata was motivated largely by the study of control circuits and computer hardware in the fifties and early sixties. Implementation of finite automata was mainly a hardware issue then.

Since the mid and late sixties, finite automata have been widely used in lexical analysis, string matching, etc. They have been implemented in software rather than hardware. However, the sizes of the automata in those applications are small in general.

Recently, finite automata and their variants have been used in many new software applications. Examples are statecharts in object-oriented modeling and design [11,12,16,9], weighted automata in image compression [6], and synchronization expressions and languages in concurrent programming languages [10]. Many of those new applications require automata of a very large number of states. For example, concatenation is a required operation in most of those applications. Consider the concatenation of two deterministic finite automata (DFA)

* This research is supported by the Natural Sciences and Engineering Research Council of Canada grants OGP0041630.

J.-M. Champarnaud, D. Maurel, D. Ziadi (Eds.): WIA'98, LNCS 1660, pp. 69–81, 1999.

with 10 states and 20 states, respectively. The resulting DFA may contain about 20 million states in the worst case [21].

It is clear that implementing finite automata in hardware is different from that in software. Hardware implementation is efficient, but suitable only for predefined automata. Adopting hardware implementation methods in software implementations is not immediate. It has to solve at least the following problems: (1) How do we represent and store a combinational network in a program efficiently in both space and time? (A table would be too big.) (2) A basic access unit in a program is a word. The access is parallel within a word and sequential among words. It would not be efficient if an implementation algorithm does not consider the word boundary and make use of it.

Implementing a small finite automaton is also different from implementing a very large finite automaton. A small finite automaton can be implemented by a word- based table or even a case or switch statement. However, these methods would not be suitable for implementing a DFA of 20 million states.

In [17], we introduced a bit-wise representation of r-AFA, which greatly improved the space efficiency in representing regular languages. We also described our algorithms and implementation methods for the union, intersection, and complementation of r-AFA.

It has been shown that a language L is accepted by an n-state DFA if and only if the reversal of L, i.e., L^R, is accepted by a $\log n$-state AFA. So, the use of r-AFA (reversed AFA) instead of DFA guarantees a logarithmic reduction in the number of states. However, the boolean expressions that are associated with each state can be of exponential size in the number of states. In our previous paper [17], we introduced a bit-wise representation for r-AFA and described the transformations between DFA and r-AFA, and also the algorithms for the union, intersection, and complementation for r-AFA. The model of r-AFA is naturally suited for bit-wise representations. NFA and DFA could also be represented in certain bit-wise forms which would save space. However, their operations would be awkward and extremely inefficient. Our experiments have shown that the use r-AFA instead of DFA for implementing regular languages can significantly improve, on average, both the space efficiency and the time efficiency for the union, intersection, and complementation operations.

We know that the resulting DFAs of the reversal and star operations of an n- state DFA have 2^n and $2^{n-1} + 2^{n-2}$ states, respectively, in the worst case, and the result of a concatenation of an m-state DFA and an n-state DFA is an $m2^n - 2^{n-1}$-state DFA in the worst case [21]. Therefore, in the worst case, the resulting r-AFA of the corresponding operations of r-AFA have basically the same state complexities, respectively. Direct constructions for the reversal, star, and concatenation of r-AFA, as described for AFA in [8,7], would have an exponential expansion in the number of states for each of the above mentioned operations.

In this paper, we present our new algorithms for the reversal, star, and concatenation operations of r-AFA. These algorithms simplify the r-AFA during the

operations. They do not necessarily produce a minimum r-AFA, but they reduce the number of states tremendously in the average case.

Our experiments show that the algorithms reduce not only the size of the state set but also the total size of a resulting r-AFA in the average case.

At the end, we also show how we have improved the DFA-to-AFA transformation algorithm described in [17]. The average run time of this transformation using the improved algorithm has been reduced significantly (by 97 percent), and for the same input DFA, the resulting AFA of the original and the improved algorithms are the same up to a permutation of states.

2 Preliminaries

The concept of alternating finite automata (AFA) was introduced in [3] and [2] at the same period of time under different names. A more detailed treatment of AFA operations can be found in [8]. In the paper [17], we modified the definition of AFA and introduced h- AFA and r-AFA. The notion of "reversed" AFA, r-AFA, is considered also in [21] but the definition there differs in a minor technical detail from the definition used here and in [17]. Below we briefly recall the definition of AFA. For a more detailed exposition and examples the reader is referred to [21]. Background on finite automata in general can be found in [19].

We denote by B the two-element Boolean algebra and B^Q stands for the set of all functions from the set Q to B.

An h-AFA A is a quintuple (Q, Σ, g, h, F), where Q is the finite set of states, Σ is the input alphabet,

$$g : Q \times \Sigma \times B^Q \to B$$

is the transition function,

$$h : B^Q \to B$$

is the accepting Boolean function, and $F \subseteq Q$ is the set of final states.

We use $g_q : \Sigma \times B^Q \to B$ to denote that g is restricted to state q, i.e., $g_q(a, u) = g(q, a, u)$, for $a \in \Sigma$, $u \in B^Q$, $q \in Q$.

We also use g_Q to denote the function from $\Sigma \times B^Q$ to B^Q that is obtained by combining the functions g_q, $q \in Q$, i.e.,

$$g_Q \equiv (g_q)_{q \in Q}.$$

We will write g instead of g_Q whenever there is no confusion.

Let $u \in B^Q$. We use $u(q)$ or u_q, $q \in Q$, to denote the component of the vector u indexed by q. By $\mathbf{0}$ we denote the constant zero-vector in B^Q (when Q is understood from the context).

We extend the definition of g of an h-AFA to a function: $Q \times \Sigma^* \times B^Q \to B$ as follows:

$$g(q, \lambda, u) = u_q$$
$$g(q, a\omega, u) = g(q, a, g(\omega, u))$$

for all $q \in Q, a \in \Sigma, \omega \in \Sigma^*, u \in B^Q$.

Similarly, the function g_Q, or simply g, is extended to a function $\Sigma^* \times B^Q \to B^Q$.

Given an h-AFA $A = (Q, \Sigma, g, h, F)$, for $w \in \Sigma^*$, w is accepted by A if and only if $h(g(w, f)) = 1$, where f is the characteristic vector of F, i.e., $f_q = 1$ iff $q \in F$.

An r-AFA A is an h-AFA such that for each $w \in \Sigma^*$, w is accepted by A if and only if $h(g(w^R, f)) = 1$, where f is the characteristic vector of F.

The transition functions of h-AFA and r-AFA are denoted by Boolean functions. Every Boolean function can be written as a disjunction of conjunctions of Boolean variables. We call a conjunction of Boolean variables or a constant Boolean function a term. A term $x_1 \wedge \ldots \wedge x_k$ is denoted simply as $x_1 \cdots x_k$. The negation of a variable x is denoted as $\bar{x}$.

The following result, that was proved in [17] states that any Boolean term can be represented by two bit-wise vectors.

Theorem 1 *For any Boolean function f of n variables that can be expressed as a single term, there exist two n-bit vectors α and β such that*

$$f(u) = 1 \Longleftrightarrow (\alpha \& u) \uparrow \beta = \mathbf{0}, \quad \textit{for all } u \in B^n,$$

where $\&$ is the bit-wise AND operator, $\uparrow$ the bit-wise EXCLUSIVE-OR operator, and $\mathbf{0}$ is the zero vector $(0, \ldots, 0)$ in B^n.

Each n-bit vector $v = (v_1, \ldots, v_n)$, $n \leq 32$ (32-bit is the normal size of a word), can be represented as an integer

$$I_v = \sum_{i}^{n} v_i 2^{i-1}.$$

We can also transform an integer I_v back to a 32-bit vector v in the following way:

$$v_i = (I_v \& 2^{i-1})/2^{i-1}, \quad 1 \leq i \leq n.$$

So, a Boolean function, which is in disjunctive normal form, can be represented as a list of terms, while each term is represented by two integers.

For an r-AFA $A = (Q, \Sigma, g, h, F)$, where $Q = \{q_1, \ldots, q_n\}$ and $\Sigma = \{a_1, \ldots, a_m\}$, we can represent g as a table of functions of size $n \times m$ with (i, j) entry corresponding to the function $g_{q_i}(a_j) : B^Q \to B$ defined by $[g_{q_i}(a_j)](u) = g_{q_i}(a_j, u)$, for $q_i \in Q$, $a_j \in \Sigma$, and $u \in B^Q$. The accepting function h can be represented as a list of integer pairs. Finally, F can be represented as a bit-vector (an integer), i.e., as its characteristic vector.

3 Algorithms for Operations of r-AFA

In this section, we present the algorithms for constructing the star and reversal of a given r-AFA and the concatenation of two given r-AFA. Note that after each construction, a simplification algorithm is applied to each Boolean function in

order to reduce the size of the Boolean expression. An algorithm which we have implemented can be found in [20]. We omit the proofs of the correctness of the algorithms due to the limit on the size of the paper.

3.1 The Star Operation

First, we consider the star operation. The algorithm eliminates all the useless states during the construction. Our experiments suggest that the r-AFA resulting from this method are significantly smaller than the naive algorithm described in [8].

Let $A = (Q, \Sigma, g, h, F)$ be a given r-AFA with $Q = \{q_1, \ldots, q_n\}$. We construct a r-AFA $A' = (Q', \Sigma, g', h', F')$ such that $L(A') = L(A)^*$ in the following. Let f be the characteristic vector of F.

Now we describe the algorithm, which deletes all the useless states during the construction.

If $n = 0$ and $h = 1$, then the r-AFA A accepts Σ^*. So, $L(A') = (L(A))^*$ is the same as $L(A)$ and we can just let $A' = A$. If $n = 0$ and $h = 0$, then A accepts the empty language and A' accepts the language that contains only the empty word λ. Then A' is the one-state r-AFA where $Q' = \{0\}$, $F' = \emptyset$, $h(x_0) = \bar{x_0}$, and $g(a, x_0) = 1$ for all $a \in \Sigma$.

Now we assume that n is a positive integer.

$F' = \emptyset$.
Let I be an array of integers of size 2^n.
Initialize $I[k] = 0$, for $k = 0, \ldots, 2^n - 1$.
Use the procedure Markarray(I, A) described at the end of this subsection to mark I, so that
$I[k] = 1 \Longleftrightarrow \exists\, x \in \Sigma^*,\ \ s.t.\ \ g(x, f) = k\ \ $, for $k \neq f$;
$I[f] = 1 \Longleftrightarrow \exists\, x \in \Sigma^*\ \ s.t.\ \ h(g(x, f)) = 1$.
If $I[f] = 0$, that means that A accepts nothing. We can construct A' the same way as in the case $n = 0$. Otherwise go to the next steps.
Let $I[k_0], \ldots, I[k_p]$ be all the entries of I that have a value 1. Let P be an array of integers of size p, such that $P[i] = k_i$.
Set $Q' = \{0, \ldots, p\}$.
Similarly as above we assume that the set $\{0, 1, \ldots, 2^n - 1\}$ is identified with B^Q. This means that, for all $i \in Q'$, the entry $P[i]$ will represent an element of B^Q.
Define the head function h' as follows:

$$h'(u) = 1 \Longleftrightarrow u = \mathbf{0} \quad \text{or} \quad \exists\, t \in Q' \ \text{ such that } \ \ u_t = 1,\ \ h(P[t]) = 1.$$

Thus,

$$h'((x_0, \ldots, x_p)) = \bar{x}_0 \ldots \bar{x}_p \vee \bigvee_{h(P[i])=1} x_i.$$

For any $a \in \Sigma$ and k such that $P[k] \neq f$, define g'_k as:

$$g'(k,a,u) = 1 \iff \exists\, s \in Q', \quad \text{such that} \quad u_s = 1,\ \ g(a,P[s]) = P[k]$$

if $u \neq \mathbf{0}$; and

$$g'(k,a,\mathbf{0}) = 1 \iff g(a,f) = P[k].$$

That is,

$$g'(k,a,(x_0,\ldots,x_p)) = \begin{cases} \bigvee_{g(a,P[i])=P[k]} x_i \bigvee \bar{x}_0 \ldots \bar{x}_p & \text{if } g(a,f) = P[k] \\ \bigvee_{g(a,P[i])=P[k]} x_i & \text{otherwise.} \end{cases}$$

Assume t is an index such that $P[t] = f$. Define g'_t as:

$$g'(t,a,u) = 1 \iff \exists\, s \in Q' \quad \text{such that} \quad u_s = 1,\ \ g(a,P[s]) = f$$
$$\text{or} \quad \exists\, r \in Q', \quad \text{such that} \quad u_r = 1,\ \ h(g(a,P[r])) = 1$$

if $u \neq \mathbf{0}$; and

$$g'(t,a,\mathbf{0}) = 1 \iff g(a,f) = f \quad \text{or} \quad h(g(a,f)) = 1.$$

Thus, $g'(t,a,(x_1,\ldots,x_p)) =$

$$\begin{cases} \bigvee_{g(a,P[i])=f \ \ or \ \ h(g(a,P[i]))=1} x_i \bigvee \bar{x}_0 \ldots \bar{x}_p & \text{if } h(g(a,f))=1 \text{ or } g(a,f)=f \\ \bigvee_{g(a,P[i])=f \ \ or \ \ h(g(a,P[i]))=1} x_i & \text{otherwise.} \end{cases}$$

The following procedure can be used to mark the array used in the above algorithm.

Procedure Markarray(I, A)

Qu is a queue of integers. Initially, Qu has only one entry f.

```
if (h(f) == 1)
    I[f] = 1;
while (!Empty(Qu)
    {
    int tmp = pop (Qu);
    int vector;
    for (a ∈ Σ)
        {
        vector = g(a, tmp);
        if(I[vector] == 0)
            {
            I[vector] = 1;
            push(vector, Qu);
            }
        if (I[f] == 0 && h(vector) == 1)
            I[f] = 1;
        }
    }
```

3.2 Concatenation of *r*-AFA

Let $A^{(i)} = (Q^{(i)}, \Sigma^{(i)}, g^{(i)}, h^{(i)}, F^{(i)})$, where $i = 1, 2$, be two r-AFAs. We construct a r-AFA $A = (Q, \Sigma, g, h, F)$ which accepts the concatenation of $L(A^{(1)})$ and $L(A^{(2)})$.

Let the numbers of states of $A^{(1)}$ and $A^{(2)}$ are n and m, and f_i be the characteristic vectors of $F^{(i)}$ for $i = 1,\ 2$ respectively. We construct A in three cases as follows:

First, let us assume $mn \neq 0$ and $\Sigma^{(1)} = \Sigma^{(2)}$.

$Q = \{q_0, \ldots, q_{n-1}, q_n, \ldots, q_{n+2^m-1}\}$, where $Q^{(1)} = \{q_0, \ldots, q_{n-1}\}$ and q_k are new states for $k \geq n$.

$$F = \begin{cases} F^{(1)} & \text{if } h^{(1)}(f_1) = 0 \\ F^{(1)} \cup \{q_{n+f_2}\} & \text{otherwise.} \end{cases}$$

We identify the numbers $n, \ldots, n + 2^m - 1$ with elements of $B^{Q^{(2)}}$, thus in the above definition of F the notation $n+f_2$ stands for the number belonging to $\{n, \ldots, n + 2^m - 1\}$ that denotes f_2.
Define $h(u) = 1 \iff \exists x,$ such that $0 \leq x \leq 2^m - 1$, $u_{x+n} = 1$, and $h^{(2)}(x) = 1$, for $u \in B^Q$; that is,

$$h((x_0, \ldots, x_{n+2^m-1})) = \bigvee_{h^{(2)}(i)=1} x_{n+i}.$$

We define $g(a, u)|_{Q^{(1)}} = g^{(1)}(a, u|_{Q^{(1)}})$, for $u \in B^Q$ and $a \in \Sigma$. That is,

$$g(q_i, a, (x_0, \ldots, x_{n+2^m-1})) = g^{(1)}(q_i, a, (x_0, \ldots, x_{n-1}))\ \ (\forall i < n).$$

Also define $g(a, u)_{q_x} = 1 \iff \exists\ y,$ such that $0 \leq y \leq 2^m - 1$ and $u_{y+n} = 1$, $g^{(2)}(a, y) = x - n$, for $x \geq n$, $x \neq n + f_2$, $u \in B^Q$ and $a \in \Sigma$; that is,

$$g(q_i, a, (x_0, \ldots, x_{n+2^m-1}) = \bigvee_{g^{(2)}(a,k-n)=i-n} x_k.$$

Define $g(a, u)_{q_{n+f_2}} = 1 \iff h^{(1)}((g(a, u)|_{Q^{(1)}})) = 1$ or $\exists\ y,$ such that $y \geq n$, $u_y = 1$, and $g^{(2)}(a, y - n) = f_2$, for all $u \in B^Q$ and $a \in \Sigma$. Thus,

$$g(q_{n+f_2}, a, (x_0, \ldots, x_{n+2^m-1}) = \bigvee_{g^{(2)}(a,k-n)=f_2-n} x_k \bigvee T(a, x_0, \ldots, x_{n-1}).$$

where the Boolean function T is the resulting function obtained by substituting $g^{(1)}(q_i, a, (x_0, \ldots, x_{n-1}))$ for x_i for all $i < n$.

Secondly, if $\Sigma^{(1)} \neq \Sigma^{(2)}$, we can construct two r-AFAs $A_1 = (Q_1, \Sigma, g_1, h_1, F_1)$ and $A_2 = (Q_2, \Sigma, g_2, h_2, F_2)$, where $\Sigma = \Sigma^{(1)} \cup \Sigma^{(2)}$, such that $L(A_i) = L(A^{(i)})$ for $i = 1, 2$. For example, let assume that $\Sigma \neq \Sigma^{(1)}$. The construction of A_1 is as follows:

Construct $Q_1 = Q^{(1)} \cup \{newq\}$, where $newq$ is a new state.
Set $F_1 = F^{(1)}$.
Define h_1 as: $h_1(u) = h^{(1)}(u|_{Q^{(1)}}) \wedge \bar{u}_{newq}$ for all $u \in B^{Q_1}$.
For any $q \in Q^{(1)}$, $a \in \Sigma$ and $u \in B^{Q_1}$, define:

$$g_1(q,a,u) = \begin{cases} g^{(1)}(q,a,u|_{Q^{(1)}}) \text{ if } a \in \Sigma^{(1)} \\ 0 \qquad\qquad\qquad\quad \text{otherwise.} \end{cases}$$

For any $a \in \Sigma$ and $u \in B^{Q_1}$, we define:

$$g_1(newq,a,u) = \begin{cases} u_{newq} \text{ if } a \in \Sigma^{(1)} \\ 1 \qquad\quad \text{otherwise.} \end{cases}$$

The last case is $mn = 0$. We can assume $\Sigma^{(1)} = \Sigma^{(2)}$ in this case (otherwise we can use the above construction to convert to this case).

Assume that $n = 0$ and $m \neq 0$. We omit the other cases here because they use a similar construction.

If $h^{(1)} = 0$, then $L(A^{(1)}) = \emptyset$. So, $L(A^{(1)})L(A^{(2)}) = \emptyset$. Therefore, we can let $A = A^{(1)}$
If $h^{(1)} = 1$, construct $A_1 = (Q_1, \Sigma, g_1, h_1, F_1)$, such that $|Q_1| = 1$ and $L(A_1) = L(A^{(1)})$ as follows:
$Q_1 = \{q\}$, $h_1 = 1$ and $F_1 = \emptyset$, where q is a new state;
$g_1(q,a,u) = 0$ for all $a \in \Sigma$, $u \in B^{Q_1}$.
Use the algorithm for the case $mn \neq 0$ and $\Sigma^{(1)} = \Sigma^{(2)}$ to construct the r-AFA accepting the concatenation of $L(A_1)$ and $L(A^{(2)})$.

Note that in the construction for the first case, we don't really need all the 2^m states that resulted from $A^{(2)}$. In fact, most of the states are of no use in general. We can use a similar method as was used in the construction for the star operation to reduce the state complexity. Briefly stated, we mark all the q_i for $i \geq n$ such that $i-n$ can be reached from f_2 under the $A^{(2)}$ transitions. Then we use the marked states to do the construction. We omit the construction here because the reader can easily fill in the details.

3.3 Reversal of r-AFA

The "reversal" of a r-AFA $A = (Q, \Sigma, g, h, F)$ is a r-AFA $A' = (Q', \Sigma, g', h', F')$ which accepts the reversal of the language of A, that is, $L(A') = [L(A)]^R$. Next we give the construction for the reversal operation, which simplifies the reversal r-AFA during the construction by removing all the useless and unreachable states. We assume $n > 0$.

Let I be an array of integers of size 2^n, Qu be a queue of integers, $T[2^n]$ be an array of integer pointers, $s = |\Sigma|$ and f be the characteristic vector of F.

Initially, let
Qu be empty;
$I[i] = 0$, for all $0 \leq i < 2^n$;
$T[i] = 0$ for all $0 \leq i < 2^n$;
int $index = 0$;
int $temp$, $vector$;
Qu.push(f);
Find all vectors $v \in B^Q$ that can be reached by f, i.e., $\exists\, x \in \Sigma^*$ such that $v = g(x, f)$. The details are as follows:

```
while (!Qu.empty())
    {
    temp = Qu.pop();
    T[index] =new int[s + 1];
    T[index][0] = temp;
    I[temp] = index;
    for (a ∈ Σ)
        {
        vector = g(a, temp);
        T[index][a] = vector;
        if ((vector! = f) && (I[vector] == 0))
           Qu.push(vector);
        }
    index++;
    }
for (int i = 0; i < index; i++)
    for (a ∈ Σ)
        T[i][a] = I[T[i][a]];
```

Construct an inverse table for T.
Let $R[index][s]$ be a two-dimensional array of sets of integers. The function Add($R[i][j], k$) add the integer k into the set $R[i][j]$. Initially, $R[i][j] = \emptyset$ for all $0 \leq i < index$, $0 \leq j < s$. Qu is also initialized to be empty.

```
for (int i = 0; i < index; i++)
    {
    for (a ∈ Σ)
        Add(R[T[i][a]][a], i);
    if (h(T[i][0]) == 1)
        {
        Qu.push(i);
        Add(F'', i);        F'' is a set of integers
        }
    }
```

Mark the new useful states.

```
Initialize I[k] to 0, for k = 0, ... , index − 1
if (h(f) == 1)    I[0] = 1;
while (!Qu.empty())
```

```
        {
        temp = Qu.pop();
        if (!empty(R, temp))
            {
            I[temp] = 1;
            for (a ∈ Σ)
               for (q ∈ R[temp][a])
                    if (I[q] == 0)
                       Qu.push(q);
            }
        }
Rename the states.
    if (I[0] == 0)
        A′ accepts the empty language;
    else
        {
        int k = 0;
        for (int i = 0; i < index; i++)
            if (I[i]! = 0)
                {
                I[i] = k;
                k++;
                }
        }
Construct A′.
    Construct F′.
        Let F′ = ∅;
        for (i in F″)
            if (I[i]! = 0)      Add(F′, I[i])
    Construct the transition function g′.
        for (int i = 0; i < index; i++)
            if (I[i]! = 0)
                for (a ∈ Σ)
                    g′(I[i], a) = 0 ⋁_{q∈R[i][a]} x_{I[q]};
    Construct the head function h′.
        h′(u) = u_0;
```

4 Improved Implementation of the DFA to r-AFA Transformation

In [17] it is shown that the bitwise representation of r-AFA significantly reduces the space needed to implement regular languages. However, manipulating bit-vectors is time-consuming if they are handled as arrays of 1's and 0's rather than integers. This is exemplified by the improvements made recently to the implementation of the DFA to r-AFA transformation of [20]. The majority of

these improvements involve increasing the efficiency of bit-vector handling, and are described below. Overall the run-time of this transformation has been decreased by 97 percent. Thus, the computation of r-AFA for large input DFA is now entirely feasible with respect to time.

4.1 Handling Bit-Vectors: Weight

The computation of an r-AFA for a given 2^n-state DFA involves the set of bit-vectors $B^n = \{0, 1, 2, .., (2^n) - 1\}$. Several parts of the DFA to r-AFA transformation involve information related to the number of 1's contained in these bit-vectors (referred to as their "weight" in [20]):

Step 2a): given an interval of integers, find the integer of lowest weight;

Step 3b): sort arrays P_f and P_n of bit-vectors ascending on weight;

Step 7 : simplifying Boolean functions; two terms t1 and t2 differ only in the negation of of one variable iff $t1 \uparrow t2$ ($\uparrow$: Exclusive OR) has weight one.

These three parts of the transformation were in fact the most time-consuming, because computing the weight of elements of B^n was done directly:

```
for(i=0; i<32; i++)
if( bit_vector & pow(2,i) == pow(2,i) )
weight++;
```

Since the transformation requires such weight-related information, it has proven extremely useful to build an array W which contains the elements of B^n in order of increasing weight. The procedure Build_Weight_Array(int n) described below accomplishes this (in time O(n)). Traversing W is then the same as traversing B^n in order of increasing weight. Thus, steps 3a) and 3b) of the transformation can be implemented as one loop:

```
for each bit-vector u in W
if( h(u)==1 )
add u to array Pf;
else
add u to array Pn;
```

After filling P_f and P_n this way, they are already sorted ascending on weight (making implementation of step 3b) unnecessary). This improvement alone reduced the runtime of the filter by 70 percent.

To compare the weights of some subset S of B^n, an array W', the inverse of W, is useful, where W'[W[i]] = i for all i in B^n. Then for any k in B^n W'[k] is the array index of k in W. So if k is the least element in set W'[S], then W[k] is the element of S with the smallest weight. Using this method to compute step 2a) also greatly reduced the overall runtime of the transformation.

Finally, one of the tests most often applied in the Boolean function simplification procedure being used is whether two terms differ only in the negation of one variable. And since we are representing terms using bit-vectors, two terms t1 and t2 differ only in the negation of one variable iff $t1 \uparrow t2$ has weight one.

Previously, all 32 positions of the bit-vector $t1 \uparrow t2$ were scanned for 1's. But if array W' is available, then

```
(the weight of bit-vector v is 1) iff (1 <= W'[v] <= n)
```

making only two integer comparisons necessary for the test. This improvement decreased the runtime of the simplification routine by 95 percent.

What follows is the pseudocode describing how array W is constructed. To simplify the description, W is described in terms of blocks, where block i contains the integers of B^n with weight i. Numbers in block $i+1$ are generated by either incrementing (giving an "incremented_int") certain numbers in block i, or shifting (giving a "shifted_int") certain numbers in block $i+1$ as follows:

```
Build_Weight_Array(n)
{
W[0] = 0;   //0 is considered a shifted_int

for(i = 1 .. n) {
for each shifted_int s in block i-1,
(next incremented_int of block i of W) := s + 1;

for each incremented_int j in block i
while( k=leftshift(j) <= (2^n)-1 )
(next shifted_int of block i of W) := k;
}
}
```

For example, let n=4. Then,

```
%
    block 0 : W[0] = ~  = 0 = 0000%
    block 1 : W[1] = 0+1    = 1 = 0001%
    W[2] = <<1  = 2 = 0010%
    W[3] = <<(<<1)   = 4 = 0100%
    W[4] = <<(<<(<<1)) = 8 = 1000%
    block 2 : W[5] = 2+1    = 3 = 0011%
    W[6] = 4+1  = 5 = 0101%
    W[7] = 8+1  = 9 = 1001%
    W[8] = <<3  = 6 = 0110%
    W[9] = <<(<<3)  = 12 = 1100%
    W[10]= <<5  = 10 = 1010%
    block 3 : W[11]= 6+1    = 7 = 0111%
    W[12]= 12+1 = 13 = 1101%
    W[13]= 10+1 = 11 = 1011%
    W[14]= <<7  = 14 = 1110%
    block 4 : W[15]= 14+1   = 15 = 1111%
```

References

1. J. Berstel and M. Morcrette, "Compact representation of patterns by finite automata", *Pixim 89: L'Image Numérique à Paris*, André Gagalowicz, ed., Hermes, Paris, 1989, pp.387-395.
2. J.A. Brzozowski and E. Leiss, "On Equations for Regular Languages, Finite Automata, and Sequential Networks", *Theoretical Computer Science* 10 (1980) 19-35.
3. A.K. Chandra, D.C. Kozen, L.J. Stockmeyer, "Alternation", *Journal of the ACM* 28 (1981) 114-133.
4. H.K. Cheung, *An Efficient Implementation Method for Alternating Finite Automata*, MSc Project Paper, Dept. of Computer Science, Univ. of Western Ontario, Sept. 1996.
5. Olivier Coudert, "Two-Level Logic Minimization: An Overview", Integration, *The VLSI Journal* 17 (1994) 97-140.
6. K. Culik II and J. Kari, "Image Compression Using Weighted Finite Automata", *Computer and Graphics*, vol. 17, 3, (1993) 305-313.
7. A.Fellah, *Alternating Finite Automata and Related Problems.* PhD dissertation, Kent State Univ. 1991.
8. A.Fellah, H.Jürgensen, and S.Yu, "Constructions for Alternating Finite Automata", *Intern J. Comp. Math.* 35 (1990) 117-132.
9. M.Fowler and K.Scott, *UML Distilled*, Addison-Wesley, 1997.
10. L. Guo, K. Salomaa, and S. Yu, "Synchronization Expressions and Languages", *Proceedings of the Sixth IEEE Symposium on Parallel and Distributed Processing* (1994) 257-264
11. D. Harel, "Executable Object Modeling with Statecharts", July 1997, *IEEE Computer*, pps. 31-42.
12. D. Harel, "Statecharts: a visual formalism for complex systems", *Science of Computer Programming* 8 (1987) 231-274.
13. H.B. Hunt, D.J. Rosenkrantz, and T.G. Szymanski, "On the Equivalence, Containment, and Covering Problems for the Regular and Context-Free Languages", *Journal of Computer and System Sciences* 12 (1976) 222-268.
14. T. Jiang and B. Ravikumar, "Minimal NFA Problems are Hard", *SIAM Journal on Computing* 22 (1993) 1117-1141.
15. D. Raymond and D. Wood, *Release Notes for Grail Version 2.5*, Dept. of Computer Science, Univ. of Western Ontario, 1996.
16. J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, *Object-Oriented Modeling and Design*, Prentice-Hall, 1991.
17. K. Salomaa, X. Wu and S. Yu, "Efficient Implementation of Regular Languages Using R-AFA", *Proceedings of the Second International Workshop on Implementing Automata. Lecture Notes in Computer Science* 1436, Springer, pps.176-184.
18. L. Stockmeyer and A. Meyer, "Word problems requiring exponential time (preliminary report)", *Proceedings of the 5th ACM Symposium on Theory of Computing*, (1973) 1-9.
19. D. Wood, *Theory of Computation*, John Wiley & Sons, New York, 1987.
20. X. Wu, "Implementation of Regular Languages by Using R-AFA", *Master's Project Report*, The Department of Computer Science, Univ. of Western Ontario, 1997.
21. S. Yu, "Regular Languages", Chapter 2, *Handbook of Formal Languages*, Vol. 1, Springer 1997.