Modular Minimization of Deterministic Finite-State Machines

Doron Bustan and Orna Grumberg*
Computer Science Department
Technion, Haifa 32000, Israel
email: {orna,doron2}@cs.technion.ac.il

Abstract

This work presents a modular technique for minimizing a deterministic finite-state machine (FSM) while preserving its equivalence to the original system. Being modular, the minimization technique should consume less time and space. Preserving equivalence, the resulting minimized model can be employed in both temporal logic model checking and sequential equivalence checking, thus reducing their time and space consumption.

As deterministic FSMs are commonly used for modeling hardware designs, our approach is suitable for formal verification of such designs.

We develop a new BDD framework for the representation and manipulation of functions and show how our minimization technique can be effectively implemented within this framework.

1 Introduction

Due to the fast development of the hardware and software industry, there is a growing need for formal verification tools and techniques. Two widely used formal verification methods are temporal logic model checking and sequential equivalence checking. Temporal logic model checking is a method for verifying finite-state systems with respect to propositional temporal logic specifications. In sequential equivalence checking, two sequential hardware designs are compared for language equivalence, meaning that for every sequence of inputs, the two designs produce the same sequence of outputs. Both model checking and equivalence checking are fully automatic. However, they both suffer from the *state explosion problem*, that is to say, their space requirements are high and limit their applicability to large systems.

Many approaches for overcoming the state explosion problem have been suggested, including abstraction, partial order reduction, modular verification methods, and symmetry [5]. All are aimed at reducing the size of the model to which formal verification methods are applied, thus extending their applicability to larger systems. When reduction methods are applied, the verification technique has to be able to deduce properties of the system by verifying the reduced model. We therefore require the result of the reduction to be *equivalent* to the original model.

Two of the most commonly used equivalence relations are *language equivalence* and *bisimulation* [15]. The former is suitable for equivalence checking as well as model checking for the

*This research has been partially supported by the Fund for the Promotion of Research at the Technion and by the Galileo Technology grant.

linear-time logic LTL /citePnueli81LTL. The latter is suitable for model checking of the expressive μ-calculus [11] logic and its widely used sublogics CTL [3, 7] and LTL.

Minimizing a model with respect to language equivalence is PSPACE-complete [16] while minimizing a model with respect to bisimulation is polynomial. Thus, bisimulation minimization appears to be tractable. However, computing bisimulation minimization in a naive way may still be quite costly in terms of time and space. This motivated the development of more subtle reduction methods for a variation of equivalence relations. We describe some of these works below.

The algorithm in [12] minimizes models with respect to bisimulation. In order to gain efficiency, the algorithm refers only to reachable states and computes equivalence classes for bisimulation instead of pairs of equivalent states. This appears to consume less memory for BDD-based [4] implementations. In [8], the algorithm presented in [12] is applied to the intersection of the model with an automaton representing the the property that should be satisfied by the model. In [6], a reduction with respect to symmetry equivalence is performed. The symmetry equivalence is a bisimulation equivalence, but not necessarily the maximal one. [6] reports that computing this reduction is more efficient in the BDD framework than reduction with respect to bisimulation.

Other works exploit modularity for reduction. The modular reduction in [1] preserves a given formula which should be checked for truth in the model. This method can result in a small model, however, since it preserves a single formula, it cannot be used for equivalence checking. In [2], the equivalence relation is a combination of language equivalence and fairness constraints. Since computing this relation is PSPACE-complete, an approximation equivalence relation is computed and the quotient model is defined with respect to the approximation equivalence relation. [9] presents an algorithm that constructs an abstract model of a system through a sequence of approximations, where the final approximation is equivalent to the original system with respect to the specification language. The approximations are constructed according to *interface specifications* which are given by the user.

1.1 Our Approach

Our approach is based on three main ideas: We restrict our attention to deterministic systems; our minimization algorithm is modular, reducing modules of the system in separation; and we develop an extension of the BDD framework that efficiently handles functions and operations on them.

Deterministic systems:

The advantages of dealing with deterministic systems are:

- Bisimulation and language equivalence coincide on deterministic systems. Thus, bisimulation minimization results in a minimal model both for language equivalence and for bisimulation.
- The algorithm presented in [10] can easily be adapted to compute bisimulation minimization.
- The transition relation of a deterministic system is actually a function, thus can be concisely represented in our new BDD framework.

Since most hardware systems are deterministic, it is beneficial to develop an efficient minimization technique for such systems. Our technique enlarges the scope of industrial-size hardware designs that can be formally verified.
Modular minimization:
The modular minimization technique is based on the partition of the system into components. Our technique minimizes the model in steps. In each step two components are selected, and their minimized composition is constructed, without ever constructing their full non-minimized composition. This process is repeated until all components are composed to form the full minimized system. The advantages of this approach are:

- Time and space requirements of minimization algorithms depend on the size of the model to which they are applied. By minimizing components instead of the full system, we expect a better overall complexity. Moreover, we will be able to minimize a system in parts even when minimizing the full system is intractable due to its size.

- It is sometimes impossible to complete the construction of the minimized system due to the size of intermediate components. In such cases, it might still be possible to apply some formal verification procedures to a partially minimized model. The partially minimized model can then be constructed by composing minimized components with unminimized ones.

BDD framework for functions:
Let $f : D \rightarrow D'$ be a complete function in which D and D' are encoded by n and n' boolean variables, respectively. Our new BDD framework represents such a function by n' boolean functions over n variables, each defining one bit in the result. Standard operations on functions such as `image`, `inverse` and `composition` are defined for this representation.

As mention above, for deterministic models the transition relation is a function. In addition, we refer to the equivalence relation as a function from states to the equivalence class they belong to. Consequently, the main steps of the minimization algorithm (presented in Section 3) can be efficiently implemented with our operations on functions. The advantages of our BDD framework are:

- The transition relation, that is typically represented by $n + n'$ boolean variables is now represented by n variables only. This is significant since BDD size often depends on the number of its variables.

- In relational representation of the transition relation, the BDD often contains unnecessary dependencies between variables. These dependencies are eliminated when the relation is represented as a set of functions. Less dependencies results in a smaller BDD. Thus, the BDD size is reduced.

The rest of the paper is organized as follows: In Section 2 we define the model, model composition and bisimulation equivalence. Section 3 describes our algorithm for minimizing a single model. Section 4 presents the modular minimization algorithm and Section 5 describes the BDD framework. Finally, in Section 6 we discuss future work.

2 Basic Definitions

We model systems as finite-state machines (FSMs) in the form of *Moore machines* in which the states are labeled with outputs and the edges are labeled with inputs. Such machines are commonly used for modeling hardware designs.

Definition 2.1 *[14] An FSM is a tuple $M =< S, S_0, I, O, L, R >$ where*

- *S is a finite set of states.*
- *$S_0 \subseteq S$ is a set of initial states.*
- *$I \cap O = \emptyset$.*
- *I is a finite set of input propositions.*
- *O is a finite set of output propositions.*
- *L is a function that maps each state to the set of output propositions true in that state.*
- *$R \subseteq S \times 2^I \times S$ is the transition relation. We assume that for every $s \in S$ and $i \subseteq I$ there exists at least one state s' such that $(s, i, s') \in R$.*

An FSM is *deterministic* iff for every state s and $i \subseteq I$ there exists exactly one state s' such that $(s, i, s') \in R$.

Two FSMs are composed only if their outputs are disjoint. There is a transition from a pair of states in the composed FSM if and only if the output of each state match the input on the transition leaving the other state. This models the input-output connections between the two machines.

Definition 2.2 *Let $M_1 =< S_1, S_{01}, I_1, O_1, L_1, R_1 >$ and $M_2 =< S_2, S_{02}, I_2, O_2, L_2, R_2 >$ be two FSMs such that $O_1 \cap O_2 = \emptyset$. The composition $M = M_1 || M_2 =< S, S_0, I, O, L, R >$ is an FSM such that:*

- *$S = S_1 \times S_2$.*
- *$S_0 = S_{01} \times S_{02}$.*
- *$I = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$.*
- *$O = O_1 \cup O_2$.*
- *$L((s_1, s_2)) = L_1(s_1) \cup L_2(s_2)$.*
- *$((s_1, s_2), i, (s_1', s_2')) \in R$ iff $(s_1, (i \cup L_2(s_2)) \cap I_1, s_1') \in R_1$ and $(s_2, (i \cup L_1(s_1)) \cap I_2, s_2') \in R_2$.*

Lemma 2.3 *Let M_1 and M_2 be deterministic FSMs, then the composition M of M_1 and M_2 is deterministic as well.*

We now define the basic notion of equivalence that we use in this work, namely, *bisimulation.*

Definition 2.4 *Let $M_1 =< S_1, S_{01}, I_1, O_1, L_1, R_1 >$ and $M_2 =< S_2, S_{02}, I_2, O_2, L_2, R_2 >$ be two FSMs such that $O_1 \cap O_2 \neq \emptyset$ and $I_1 = I_2$. We say that M_1 and M_2 are bisimulation equivalent with respect to $O' \subset O_1 \cap O_2$ iff there exists a relation $H \subseteq S_1 \times S_2$ (called bisimulation relation) such that:*

- *For every state $s_{01} \in S_{01}$ there exists a state $s_{02} \in S_{02}$ such that $(s_{01}, s_{02}) \in H$ and for every state $s_{02} \in S_{02}$ there exists a state $s_{01} \in S_{01}$ such that $(s_{01}, s_{02}) \in H$.*
- *For every pair (s_1, s_2) in H the following three conditions hold:*
  - $L_1(s_1) \cap O' = L_2(s_2) \cap O'$.
  - *For every $i \subseteq I_1$ (recall that $I_1 = I_2$), and for every state s_1' such that $(s_1, i, s_1') \in R_1$ there exists a state s_2' such that $(s_2, i, s_2') \in R_2$ and $(s_1', s_2') \in H$.*
  - *For every $i \subseteq I_2$, and for every state s_2' such that $(s_2, i, s_2') \in R_2$ there exists a state s_1' such that $(s_1, i, s_1') \in R_1$ and $(s_1', s_2') \in H$.*

Bisimulation is an equivalence relation over FSMs. [13] shows that for every two FSMs M_1 and M_2, there exists a maximal bisimulation relation, which contains every relation that satisfies the conditions of Definition 2.4. The maximal bisimulation relation $H \subseteq S \times S$ over the states of an FSM M is an equivalence relation over S. As such, it induces a partition of S to equivalence classes. These classes can be used to form the *quotient FSM* of M, which is the minimal FSM that is bisimulation equivalent to M. Formally,

Definition 2.5 *Let $M =< S, S_0, I, O, L, R >$ be an FSM and let $H \subseteq S \times S$ be the maximal bisimulation relation with respect to $O' \subseteq O$ over M. The quotient FSM $M_Q =< S_Q, S_{0_Q}, I_Q, O_Q, L_Q, R_Q >$ of M with respect to H is defined as follows:*

- $S_Q = \{\alpha | \alpha$ *is an equivalence class in* $H\}$.
- $S_{0_Q} = \{\alpha |$ *there exists* $s_0 \in S_0$ *such that* $s_0 \in \alpha\}$.
- $I_Q = I$.
- $O_Q = O'$.
- *For* $\alpha \in S_Q$, $L_Q(\alpha) = L(s) \cap O'$, *for some (all) states* $s \in \alpha$.
- $R_Q = \{(\alpha, i, \alpha') |$ *there are states* $s \in \alpha, s' \in \alpha'$ *such that* $(s, i, s') \in R\}$.

Definition 2.6 *An FSM M is minimized iff it is isomorphic to its quotient structure.*

Proposition 2.7 *Every quotient FSM is minimized.*

For the rest of this paper, we will use the term "minimized FSM" for quotient FSM.

Proposition 2.8 *Let M be an FSM and M_Q be the quotient FSM of M with respect to O'. Then M_Q is the smallest (in number of states and transitions) FSM which is bisimulation equivalent to M with respect to O'.*

Proposition 2.9 *If M is deterministic then its quotient FSM M_Q is deterministic as well.*

3 Minimizing Deterministic Systems

In this section we present an adaptation of the algorithm given in [10] for constructing the quotient automaton for a given regular deterministic automaton. The algorithm is adapted for deterministic FSMs, so that given an FSM, it constructs its quotient FSM.

Let $H \subseteq S \times S$ be the maximal bisimulation relation over M. Since M is deterministic, the conditions for two states to be relates by H can be slightly changed. $(s_1, s_2) \in H$ if and only if

1. $L(s_1) = L(s_2)$.
2. For every input $i \in I$, $(R(s_1, i), R(s_2, i)) \in H$.

Recall that H is an equivalence relation over S. In order to improve the performance of our minimization algorithm, we refer to H as a function from the states in S to their equivalence classes. The conditions above are now stated as:

1. $H(s_1) = H(s_2) \rightarrow L(s_1) = L(s_2)$.
2. $H(s_1) = H(s_2) \rightarrow$ for every input $i \in I$, $H(R(s_1, i)) = H(R(s_2, i))$.

The main difference between the algorithm in [10] and our algorithm is in the initial partitioning. While for automata the initial partition forms two sets (accepting and rejecting), the states of a deterministic FSM are initially partitioned into $2^{|AP|}$ sets, one for each state labeling. Our algorithm is presented in Figure 1. Next we give an intuitive explanation of how the algorithm works. We say that a class σ is *stable* with respect to a class σ' and an input i if the i-successors (successors on input i) of states in σ are either all in σ' or all outside σ'. If σ is unstable with respect to (σ', i) then (σ', i) is called a *splitter*.

Initially the states are partitioned according to their labeling. The function `initH()` creates 2^{AP} classes and initializes H so that each state s is mapped to the class with the same labeling as s.

Let Σ be the set of classes and Γ be the set of potential splitters. The algorithm works in iterations. At each iteration a potential splitter (σ, i) is chosen from Γ. For each class σ_j, if it is unstable with respect to the splitter then it is split into two. σ_j remains with the states whose i-successors are in σ. σ_k is a new class containing the states of σ_j whose i-successors are outside σ. Next, for each input i', either (σ_j, i') or (σ_k, i') is inserted to Γ.

The algorithm stops when all classes are stable with respect to all inputs and all classes. At this stage, two states are in the same class if and only if they are bisimulation equivalent.
In the algorithm we use the notation $R_i^{-1}(S')$ for $\{s | R(s, i) \in S'\}$.
Note that, the algorithm is presented in a set-based notation. This is done in order to allow a straight forward implementation with BDDs. Since both R and H are functions, the algorithm is suitable for implementation within our new BDD framework, as explained in Section 5.

4 The Modular Minimization Technique

In this section we present our modular minimization algorithm. The algorithm receives a design, given as a set of n components. It works in steps. In each step two minimized components M and M' are selected and a new minimized component is constructed, which is equivalent to $M||M'$. The algorithm terminates when a step results in a single component. In this case, the final component is the smallest in terms of states and transitions which is equivalent to the composition of the n original components.

In this section we focus on an improvement of a single step. Given two minimized components M and M', their composition $M||M'$ is not necessarily minimized. This is demonstrated in Figure 2. A naive solution might first compose M and M' and then minimize them. This, however, may result in unnecessarily large intermediate components. Thus, this solution will

```
Reduction(M){
 initH()
 $k = 2^{|AP|}$
 $\Gamma = \Sigma \times 2^I$  // set of all pairs of classes and inputs
 while ($\Gamma \neq \emptyset$) do
    select $(\sigma, i)$ from $\Gamma$
    $T' = H^{-1}(\sigma)$
    let $T = \{t | (t, i) \in R^{-1}(T')\}$
    let $\Theta = H(T)$
    while ($\Theta \neq \emptyset$) do
       select $\sigma_j$ from $\Theta$
       let $\Theta = \Theta \setminus \sigma_j$
       let $S = H^{-1}(\sigma_j)$
       let $S' = S \cap T$
       let $S'' = S \setminus S'$
       if $S'' \neq \emptyset$ and $S' \neq \emptyset$ then
          create new class $\sigma_k$
          move($S''$,$\sigma_j$ ,$\sigma_k$)  // Moves the states in $S''$ from $\sigma_j$ to $\sigma_k$
          for each $i \in I$ do
             if $((\sigma_j, i) \notin \Gamma$ and $|R_i^{-1}(S')| \leq |R_i^{-1}(S'')|$ then
                add $(\sigma_j, i)$ to $\Gamma$
             else
                add $(\sigma_k, i)$ to $\Gamma$
             endif
          endfor
          $k = k + 1$
       endif
    endwhile
 endwhile
}
```

Figure 1: The minimization algorithm

require more space than is actually needed for the result. We, on the other hand, suggest an algorithm that constructs the result without constructing the composition of the original components.

Below we present our modular minimization algorithm. The algorithm is given two minimized FSMs M_1 and M_2. We use the notation $M = M_1 || M_2$, $O_1' = O_1 \cap I_2$, and $O_2' = O_2 \cap I_1$. The algorithm performs the following steps:

1. Reduce M_1 with respect to O_1'. We call the result M_1^r.
2. Reduce M_2 with respect to O_2'. We call the result M_2^r.
3. Compose $M_1^e = M_1 || M_2^r$.
4. Compose $M_2^e = M_1^r || M_2$.
5. Reduce M_1^e with respect to O_1. We call the result M_1^d.

6. Reduce M_2^e with respect to O_2. We call the result M_2^d.

7. Compose $M_d = M_1^d || M_2^d$.

The table below presents the inputs and outputs of each FSM.

FSM	Input	Output
M_1	I_1	O_1
M_2	I_2	O_2
M_1^r	I_1	O_1'
M_2^r	I_2	O_2'
M	$(I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	$O_1 \cup O_2$
M_1^e	$(I_1 \setminus O_2') \cup (I_2 \setminus O_1) = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	$O_1 \cup O_2'$
M_2^e	$(I_1 \setminus O_2) \cup (I_2 \setminus O_1') = (I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	$O_2 \cup O_1'$
M_1^d	$(I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	O_1
M_2^d	$(I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	O_2
M_d	$(I_1 \setminus O_2) \cup (I_2 \setminus O_1)$	$O_1 \cup O_2$

An example for the *modular minimization technique* is presented in Figure 2.

The intuition behind the modular minimization algorithm is as follows. When two FSMs are composed, each restricts the behavior of the other by providing a real environment, rather than an open one. In the restricted environment, states that behave differently in the open environment are now indistinguishable and can be collapsed into the same equivalence class.

Our goal is to minimize M_1 and M_2 in separation, while taking into account the environment each runs in. While minimizing M_2 it is sufficient to consider only the part of M_1 which influences M_2. M_1^r is exactly that part. Therefore, states in M_2 that become indistinguishable in $M = M_1 || M_2$ are also indistinguishable in $M_2^e = M_1^r || M_2$. These states are collapsed, resulting in M_2^d. Similarly, in M_1^e states of M_1 that are indistinguishable in M are collapsed (resulting in M_1^d). When M_1^d and M_2^d are finally composed, the resulting FSM contains no further redundancy. Thus, no further minimization is needed.

The skeleton of the correctness proof for the algorithm is listed in the lemma below.

Lemma 4.1

- *M_1^e and M are bisimulation equivalent with respect to $O_1 \cup O_2'$.*
- *M_2^e and M are bisimulation equivalent with respect to $O_2 \cup O_1'$.*
- *M_1^d and M are bisimulation equivalent with respect to O_1.*
- *M_2^d and M are bisimulation equivalent with respect to O_2.*
- *M_d and M are bisimulation equivalent with respect to $O_1 \cup O_2$*
- *M_d is minimized with respect to $O_1 \cup O_2$.*

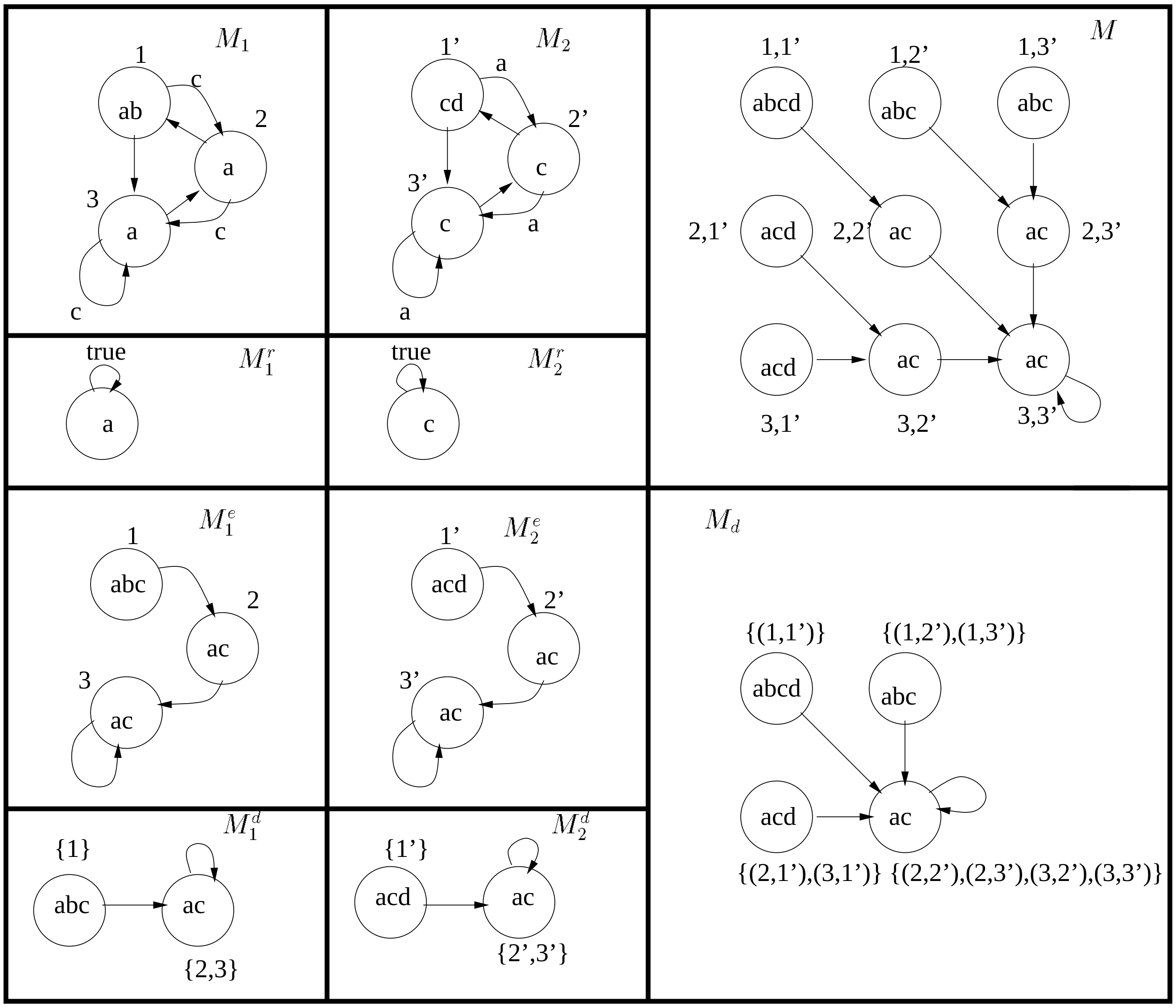

Figure 2: An example of the modular minimization algorithm: M_1 has input set $I_1 = \{c\}$ and output set $O_1 = \{a, b\}$. M_2 has input set $I_2 = \{a\}$ and output set $O_2 = \{c, d\}$. Note that, even though M_1 and M_2 are minimized, M is not. M_d is the quotient model of M. It can also be obtained by composing M_1^d and M_2^d.

4.1 Time and Space Complexity

The algorithm we present includes two basic operations:

1. Composing two FSMs $M'' = M||M'$. The most costly part in time and space of this operation is the computation of the transition relation R''. This can be done in time and space complexity of $O(|R''|)$.

2. Minimizing an FSM M into its quotient FSM M_Q. Our algorithm has the same complexity as the one in [10]. Its space complexity is $O(|R|)$ and its time complexity is $O(|R| \cdot log(|S|))$.

Thus, the minimization is the dominate part of the algorithm.

Since $|M_1| \leq |M_1^e|$ and $|M_2| \leq |M_2^e|$, the complexity of our algorithm depends on the sizes of M_1^e and M_2^e. If $|M_1^r| \ll |M_1|$ and $|M_2^r| \ll |M_2|$, then $|M_1^e| \ll |M|$ and $|M_2^e| \ll |M|$. In this case, our algorithm will have a significantly better complexity.

However, in the worst case where $|M_1^r| = |M_1|$ and $|M_2^r| = |M_2|$, $|M_1^e| = |M_2^e| = |M|$. Thus, in the worst case our algorithm has the same complexity as the naive algorithm that first composes M_1 and M_2 and then minimizes the composed FSM.

5 The BDD Framework

BDDs [4] are widely used in symbolic model checking and equivalence checking. Efficient representation of sets and relations by BDDs has been the subject of extensive research. However, no special consideration has been given to efficient representation of relations which are functions. In this section we show how to represent functions concisely using BDDs.

Let f be a function $f : D \rightarrow D'$ where $|D| = 2^n$ and $|D'| = 2^{n'}$. In order to represent f by a BDD, f is first encoded as a boolean relation over $n + n'$ boolean variables, where n variables encode the domain of f, D, and n' variables encode the range of f, D'. The usual BDD representation of such a function includes all $n + n'$ variables. Alternatively, we suggest to represent f by means of n' boolean functions over n variables, each defines the value of one variable in the encoding of D'. Since the BDDs are defined over a smaller number of variables, they are expected to be smaller in size.

5.1 Preliminaries

We describe BDDs as presented in [4]. We use $x_1, x_2, \ldots, x_n$ to denote boolean variables and $g(x_1, x_2, \ldots x_n)$ to denote a boolean function. Let $d \in \{0,1\}^n$, we use the functions $v_i(d)$ to denote the value of the i'th bit in d. Sometimes we use x_i as the function $x_i = 1$ and $\overline{x_i}$ as $x_i = 0$. A BDD is always defined with respect to an order over the variables.

Definition 5.1 *A BDD is a DAG (Directed Acyclic Graph) with one root and at most two leaves. The leaves are labeled with* $0, 1$ *and the non-leaf nodes are labeled with a variable* x_i*. Every non-leaf node has exactly two successors (low,high). If node* nd' *is a successor of node* nd *then either* nd' *is a leaf or the variable labeling node* nd' *is greater by the variable order than the one labeling node* nd*.*

A BDD B with root nd represents a boolean function g_{nd}, defined recursively as follows:

- If nd is a leaf then it represents the label of nd (0 or 1).
- If nd in non-leaf node which is labeled with variable x_i, then $g_{nd} = (x_i \wedge g_{nd.high}) \vee (\overline{x_i} \wedge g_{nd.low})$.

A BDD B representing a function g can also be viewed as representing a set $A \subseteq \{0,1\}^n$ such that $a \in \{0,1\}^n$ is an element of A if and only if $g(a) = 1$. We sometimes refer to A by *root*, the root node of B.

Definition 5.2 *A BDD B is reduced if it satisfies the followings:*

1. *There are no two different nodes in B which represent the same function.*

2. Each non-leaf node nd in B satisfies: nd.high $\neq$ nd.low.

[4] shows that for every boolean function there exists a reduced BDD that represents it; for the rest of this paper we refer only to reduced BDDs. [4] also shows that BDDs have the following property:
Let B_1 and B_2 be two BDDs representing functions g_1, g_2 respectively, such that B_1 and B_2 have the same variable order. Then $g_1 \equiv g_2$ iff $B_1 = B_2$.

In addition, [4] suggests efficient procedures that implement operations over boolean functions represented by BDDs.

In formal verification, BDDs are used to encode sets of states of the verified FSM, its transition relation and labeling function. To do so, the states of the FSM are encoded by boolean variables. The set of states is then represented by a boolean function, as described above. The transition relation, $R \subseteq S \times 2^I \times S$ is viewed as a set of triples and is represented in a similar manner.

5.2 A new BDD representation for functions

Assume that we have two finite domains D and D' such that $|D| = 2^n$ and $|D'| = 2^{n'}$ $(n, n' \in I\!N)$. Let $f : D \rightarrow D'$ be a complete function. We encode the elements of D by $x_1, x_2, \ldots x_n$ and the elements of D' by $x'_1, x'_2, \ldots, x'_{n'}$. Given an element $d \in D$, $f(d)$ is a unique element of D'. Thus the values of the variables that encode d' depend only on d. We define n' sets $f_1, f_2, \ldots, f_{n'}$ of subsets of D such that $d \in f_j \Leftrightarrow v_j(f(d)) = 1$. Another way to look at f_j is as a function $f_j : D \rightarrow \{0, 1\}$, which determine the value of x'_j. Each of these subsets can be represented by a BDD; thus we represent f as n' BDDs over $x_1, x_2, \ldots, x_n$. Next we show how to implement typical operations on functions for this form of representation.

First we present an algorithm for computing f^{-1}. The algorithm receives a BDD that represent a set Q' and construct a BDD that represents $Q = f^{-1}(Q')$. The algorithm is shown in Figure 3.

```
BDD inverse(BDD Q'){
    return inverseNode(Q'.root)
}

inverseNode(node $nd$){
    if ($nd$ is a terminal node ) then return $nd.value$
    $j = nd.index$
    return $(\overline{f_j} \cap$inverse$(nd.low)) \cup (f_j \cap$inverse$(nd.high))$
}
```

Figure 3: The inverse algorithm

In order to see why this algorithm works correctly, we need the following definitions and claims. The first proposition is immediate from the definition.
Proposition 5.3 *Given an element $d' \in D'$. An element $d \in D$ satisfies $f(d) = d'$ iff for every $1 \leq j \leq n'$, $d \in f_j \Leftrightarrow v_j(d') = 1$.*

Definition 5.4 *We define f_j^{-1} as follows:*

$$f_j^{-1}(b) = \begin{cases} f_j & b = 1 \\ \overline{f_j} & b = 0 \end{cases} .$$

The next proposition rephrases Proposition 5.3, using the notation f_j^{-1}.

Proposition 5.5 *Given an element d', the following holds: $f^{-1}(d') = \cap_{j=1}^{n'} f_j^{-1}(v_j(d'))$.*

We now extend the previous proposition to a set of states.

Corollary 5.6 *Given a subset $Q' \subseteq D'$, let $f^{-1}(Q') = \{d | \exists d'.\ d' \in Q' \text{ and } f(d) = d'\}$ then $f^{-1}(Q') = \cup_{d' \in Q'}(\cap_{j=1}^{n'} f_j^{-1}(v_j(d')))$.*

Note that the computation of $Q = f^{-1}(Q')$ as presented in the previous corollary requires to handle the elements of Q' one at a time and do not take advantage of the BDD representation of Q'. The next proposition shows how to compute Q based on the BDD representation of Q'.

The computation follows the following intuition. Suppose the set $Q' \subseteq \{0,1\}^{n'}$ is represented by a BDD B'_Q so that x_j is the variable in the root of B'_Q. Elements of Q', represented by *root.low* are those in which $x_j = 0$, thus, they will be mapped to by elements from $\overline{f_j}$. Similarly, the elements of Q', represented by *root.high* are those in which $x_j = 1$, and therefore they will be mapped to by elements from f_j.

Proposition 5.7 *Let Q' be a subset of D'. Let B'_Q be the BDD that represents Q'. Let j be the index of the root of B'_Q. Then $Q = (f_j \cap f^{-1}(root.high)) \cup (\overline{f_j} \cap f^{-1}(root.low))$.*

We next describe an algorithm that computes the image of a set Q. Figure 4 presents an algorithm that gets a BDD that represents a set $Q \subseteq D$; the algorithm constructs the BDD that represents $Q' = f(Q)$.

```
BDD image(BDD Q){
    return imageNode(Q,1)
}

BDD imageNode(BDD Q, int $j$){
    if ($j > n'$ ) then return $true$
    if ($Q = \emptyset$) return $false$
    return ($x_j \cap$imageNode($Q \cap f_j, j+1$)) $\cup$($\overline{x_j}\cap$imageNode($Q \cap \overline{f_j}, j+1$))
}
```

Figure 4: The image algorithm

Intuitively, we need to determine which elements of $\{0,1\}^{n'}$ are in $Q' = f(Q)$. This is done by determining the elements in $f(Q \cap f_j)$ (for which $x_j = 1$) and the elements in $f(Q \cap \overline{f_j})$ (for which $x_j = 0$).

While the number of BDD operations performed by the `inverse` algorithm is linear in the size of the BDD representing Q', the number of BDD operations in the `image` algorithm is linear in the size of Q'. Since the BDD representation of a set is often much smaller than the set itself, `inverse` will usually be much more efficient than `image`.

Another important operation is the composition of two functions which is defined as follows. Let D, D', D'' be domains and let $f : D \to D'$ and $g : D' \to D''$ be functions represented by $f_1, \dots f_{n'}$ and $g_1, \dots g_{n''}$ respectively. The function $h = g \circ f$ $(h : D \to D'')$ is represented by n'' BDDs. The j'th BDD represent the set $h_j = \{d | v''_j(f(g(d))) = 1\}$, and is computed by $h_j = f^{-1}(g_j)$.

Finally we show how to transform our presentation of a function into a relation like presentation, and vice versa.

Let $f_1, \dots, f_{n'}$ be the BDDs which represent f. We would like to construct a BDD that represents a relation F over $D \times D'$, such that $(d, d') \in F$ iff $f(d) = d'$. In order to represent F we use an additional set of variables $x'_1, \dots, x'_{n'}$ which is used to encode the elements of D'. We now construct F as follows $F = \cap_{j=1}^{n'} (f_j \leftrightarrow x'_j)$.

Let F be a relation over $D \times D'$ such that $(d, d') \in F$ iff $f(d) = d'$. Then f_j is computed as follows: $f_j = \exists x'_1, \dots, x'_{n'} (F \wedge x'_j)$.

5.3 An Example: Modeling Deterministic FSM by BDDs for Functions

The example in this section demonstrates how the BDD representation for functions can be used for representing an FSM. In addition, we show how to compute the set of predecessors Q for a given set of states Q'. This is a central operation in formal verification algorithms (often referred to as `preimage`).

Consider the FSM in Figure 5. Its set of states is $S = \{00, 01, 10, 11\}$. Its input set is $I = \{a\}$. The transition function $R : S \times I \to S$ is shown in the table below. In the table we use the variables (x_0, x_1) to encode S, i_0 to encode I and x'_0, x'_1 to encode the next states in the transition relation.

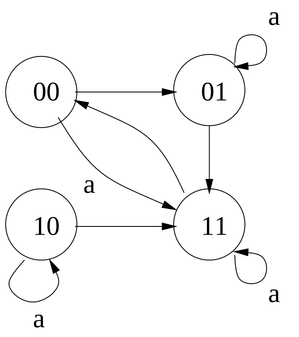

Figure 5: An example FSM

x_0	x_1	i_0	x'_0	x'_1
0	0	0	0	1
0	0	1	1	1
0	1	0	1	1
0	1	1	0	1
1	0	0	1	1
1	0	1	1	0
1	1	0	0	0
1	1	1	1	1

In order to define R is our BDD framework we partition it to two boolean functions R_0 and R_1

so that $R_0 = \{001, 010, 100, 101, 111\}$ and $R_1 = \{000, 001, 010, 011, 100, 111\}$. R_0 consists of the set of encodings of (s, i) for which $x'_0 = 1$. Similarly, R_1 is the set of encodings for which $x'_1 = 1$.

Next we show how to use this representation in order to compute $R^{-1}(Q')$ using the `inverse` algorithm. The result of this operation is the set of pairs (s, i) such that $(s, i) \in R^{-1}(Q')$. Thus, the set Q of predecessors of Q' is computed by $Q = \{s | \exists i \in I.(s, i) \in R^{-1}(Q')\}$.

Let $Q' = \{01, 10\}$. The BDD $B_{Q'}$ is shown in Figure 6. The inverse algorithm results in

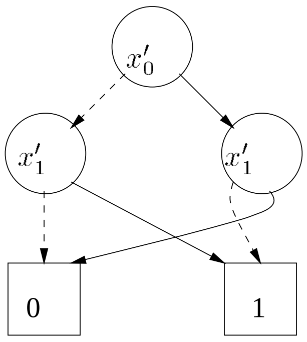

Figure 6: The BDD $B_{Q'}$. Dashed lines lead to *low* successors; full lines lead to *high* successors.

$R^{-1}(Q) = (\overline{R_0} \cap ((\overline{R_1} \cap 0) \cup (R_1 \cap 1))) \cup (R_0 \cap ((\overline{R_1} \cap 1) \cup (R_1 \cap 0))) = (\overline{R_0} \cap R_1) \cup (R_0 \cap \overline{R_1}) = \{000, 011, 101\}$. The set of predecessor is now computed by $Q = \{q | \exists i.(q, i) \in R^{-1}(Q')\} = \{00, 01, 10\}$.

6 Directions for Future Research

We are currently working on an implementation of the modular minimization technique. Our goal is to incorporate this technique into a framework for model checking and sequential equivalence checking of hardware designs. In order to do so, several additional issues should be considered. These issues may have a great influence on the effectiveness of our method.

1. Heuristics are needed in order to determine the partition of the system into components. Both the sizes of the components and the size of their input-output interface will be taken into account.

2. The order in which the components are composed should be determined. This may strongly affect the space requirements of intermediate results.

References

[1] A. Aziz, T.R. Shiple, V. Singhal, and A.L. Sangiovanni-Vincetelly. Formula-dependent equivalence for compositional CTL model checking. In D. Dill, editor, *Proceedings of the Sixth Conference on Computer Aided Verification (CAV'94)*, volume 818 of *LNCS*, pages 324–337, 1994.

[2] A. Aziz, V. Singhal, G.M. Swamy, and R.K. Brayton. Minimizing interacting finite state machines: A compositional approach to language containment. In *Proceedings of the International Conference on Computer Design*, pages 255–261, 1994.

[3] M. Ben-Ari, Z. Manna, and A. Pnueli. The temporal logic of branching time. *Acta Informatica*, 20:207–226, 1983.

[4] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35:677–691, August 1986.

[5] E.M. Clarke, O. Grumberg, and D.A. Peled. *Model Checking.* MIT Press, December 1999.

[6] E.M. Clarke, R.Enders, T. Filkorn, and S. Jha. Exploiting symmetry in temporal logic model checking. In *Formal Methods in System Design*, pages 77–104, 1996.

[7] E.A. Emerson and E.M. Clarke. Characterizing correctness properties of parallel programsusing fixpoints. In *LNCS*, volume 85, pages 169–181, 1980.

[8] K. Fisler and M. Vardi. Bisimulation minimization in an automata-theoretic verification framework. In *Formal Methods in Computer-Aided Design (FMCAD)*, pages 115–132, 1998.

[9] Susanne Graf, Bernhard Steffen, and Gerlad Lüttgen. Compositional minimisation of finite state systems using interface specifications. *Formal Aspects of Computing*, 8(5):607–616, 1996.

[10] J. E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. In Z. Kohavi and A. Paz, editors, *Theory of Machines and Computations.* Academic Press, New York, 1971.

[11] D. Kozen. Results on the propositional μ-calculus. *TCS*, 27, 1983.

[12] D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proceedings of the 24th ACM Symp. on Theory of Computing*, 1992.

[13] R. Milner. *Communication and Concurrency.* Prentice-Hall, Englewood Cliffs, New Jersey, 1989.

[14] E. F. Moore. Gedanken–experiments on sequential machines. In C. E. Shannon and J. McCarthy, editors, *Annals of Mathematics Studies (34), Automata Studies*, pages 129–153. Princeton University Press, Princeton, NJ, 1956.

[15] D. Park. Concurrency and automata on infinite sequences. In *5th GI-Conference on Theoretical Computer Science*, pages 167–183. Springer-Verlag, 1981. LNCS 104.

[16] A. Sistla, M. Vardi, and P. Wolper. The complementation problem for Buchi automata with applications to temporal logic. In *In Proc. 10th Int. Colloquium on Automata, Languages and Programming*, volume LNCS 194, pages 465–474, 1985.