IMPLICIT MANIPULATION OF EQUIVALENCE CLASSES USING BINARY DECISION DIAGRAMS

by

Bill Lin and A. Richard Newton

Memorandum No. UCB/ERL M91/13

28 January 1991

IMPLICIT MANIPULATION OF EQUIVALENCE CLASSES USING BINARY DECISION DIAGRAMS

by

Bill Lin and A. Richard Newton

Memorandum No. UCB/ERL M91/13

28 January 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

TITLE PAGE

IMPLICIT MANIPULATION OF EQUIVALENCE CLASSES USING BINARY DECISION DIAGRAMS

by

Bill Lin and A. Richard Newton

Memorandum No. UCB/ERL M91/13

28 January 1991

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Implicit Manipulation of Equivalence Classes Using Binary Decision Diagrams *

Bill Lin A. Richard Newton
Electrical Engineering and Computer Sciences
University of California, Berkeley, CA 94720

Abstract

We address the problem of manipulating equivalence classes implicitly. The relevance of this fundamental concept is shown by giving examples of applications. A new boolean operator called *compatible projection* is presented as a means for finding a compatible function corresponding to a given relation. A fundamental property of the compatible projection operator is that the function produced is canonical. In manipulating equivalence classes, the compatible projection operation implicitly derives an encoding function that encodes the equivalence class information symbolically. The main limitation is the size of the BDD representing the encoding function.

*This project was supported in part by NSF/DARPA under contract number MIP-8719546.

1 Introduction

Recently, it has become apparent that many problems in synthesis and verification are intimately related in that they are often fundamentally dependent on the same set of basic logic manipulations. Efficient techniques developed in this area can be viewed as "core" technologies that can be applied to a wide spectrum of applications in both arenas. Since many algorithms in synthesis and verification make use of the same basic set of core computations extensively, new advances in this area are extremely important.

One such core technology is the *binary decision diagram* (BDD) [3]. Although BDD's were originally developed for symbolic simulation and verification, they have been found to be fundamental in a wide variety of applications [12, 10]. In general, BDD's can be used to solve problems involving intensive boolean manipulations or computations. Another recent, and extremely important, core technology is the concept of BDD-based implicit state enumeration developed by Coudert et al. [6]. The main idea is to use BDD's to perform symbolic breadth-first execution of an implicitly defined state space. At each iteration, a large number of states is simultaneously traversed by performing symbolic image and inverse image computations. This concept can be applied to solve many problems requiring spatial or temporal analysis of some state space. These techniques are related to the concept of characteristic function. Already, it has been applied to solve problems in verification [4, 6] and synthesis [11]. In [11], implicit enumeration was applied to compute sequential don't care conditions from a gate-level datum that can be used to minimize logic during synthesis.

While BDD's and implicit enumeration provide the basic machinery to manipulate functions and relations efficiently, we do not have at our disposal analogous machinery for manipulating and representing *equivalence classes* efficiently (other than straightforward truth table enumeration). However, in many interesting applications, the ability to manipulate equivalence classes is in fact the fundamental bottleneck. For example, when analyzing a subnetwork embedded in a hierarchically defined boolean network, a number of output patterns may be considered *equivalent* with respect to the primary outputs of the global network [2]. These equivalent output patterns may be grouped together into equivalent classes for optimization. Using implicit enumeration techniques, only pairwise equivalent relationships can be derived. However, we have no way of deriving or representing the equivalence classes other than quasi-exhaustive methods. Since the number of signals through a cut of a boolean network is usually very large, this severely limits the analysis to only small problems. A related problem is computation of *communication complexity*. The problem is to compute information flow from one part of a network to the other given a 2-way partition of primary inputs. It has important applications in logic synthesis [7] and functional decomposition [8]. The only known technique to date is the use of a communication matrix [7], which degenerates to an exhaustive analysis. Another interesting problem is the analysis of sequential circuits. BDD-based algorithms exist for computing equivalent state pair information based on implicit backward traversal [11]. This equivalence relation information can be used to advantage in a number of important problems in synthesis and verification. For example, the equivalent states information can be used to state minimize large finite state machines by merging equivalent states. However, efficient techniques for manipulating equivalence classes are missing.

In this paper, we present new BDD-based techniques for manipulating equivalence classes implicitly. The techniques are intended to complement existing BDD and implicit enumeration techniques to provide a reservoir of logic manipulation machinery. These techniques are based on an important new boolean operator called the *compatible projection operator*. The compatible projection operator can be seen as a mechanism for selecting a compatible mapping corresponding to a relation. The compatible projection operator has the remarkable property that, when applied to a relation, it is guaranteed to produce a *canonical* function compatible with the relation. This result is significant since in general there may be an astronomical number of compatible mappings. In manipulating equivalence classes, the compatible projection operator is used to implicitly derive an encoding function that encodes the equivalence class information symbolically. We give in this paper examples of applications where the techniques presented have been able to analyze problems involving over 10^{68} equivalence classes. Potentially much larger problems can be handled with the real limitation on the ability to represent the function that encodes the equivalence class information in BDD form.

2 Theoretical Groundwork

2.1 Binary Decision Diagrams

Binary decision diagram (BDD) is a directed acyclic graph (DAG) representation of boolean logic [3] corresponding to a recursive Shannon decomposition. Each node is associated with a variable and two fanout arcs: one corresponds to the function when the variable is set to 0, and the other corresponds to when the variable is set to 1. At the leaves of the graph are two constant nodes representing the constants 0 and 1. A variable ordering is imposed such that all transitive fanouts of a node must have a higher ordering index, except for the constant nodes. Also, a variable may not be repeated in a path. The BDD is said to be reduced if there are no isomorphic subgraphs. This representation is canonical for a given variable ordering. Standard boolean operations like intersection, union, and negation can be implemented very efficiently with BDD's. Boolean quantifiers such as the existential and universal operators can also be implemented with BDD's as follows. The existential quantification (also called *smoothing*) of the boolean formula f with respect to a boolean variable x is

$$(\exists x)f \overset{\text{def}}{=} f_x + f_{\overline{x}},$$

where f_x is the usual cofactor operation as defined in [3]. Similarly, the universal operator (also called the *consensus* operator) can be computed as

$$(\forall x)f \overset{\text{def}}{=} f_x f_{\overline{x}}.$$

The advantage of BDD's over other canonical representations (such as a truth table) is that they are usually much more compact and efficient polynomial time algorithms exist to manipulate them.

2.2 Relations and Classes

A *relation* is a subset over the cartesian product of some, possibly non-boolean, finite domains $\mathcal{R} \subseteq D_1 \times D_2 \times \ldots \times D_n$. The *cardinality* of a finite set D is denoted by $\#D$ or $|D|$. In the remainder of the paper, we will assume the domains are over boolean values. Non-boolean domains (i.e. symbolic relations [10]) can be easily handled by using a homomorphic encoding function $\xi : D \rightarrow \mathbf{B}^k$ which maps one-to-one each element of D to a unique boolean vector of length k, where $\mathbf{B}$ is the set $\{0, 1\}$ and $k \geq \lceil \log_2 \#D \rceil$. Hence, no generality is loss. For most applications considered, the choice of encodings and code lengths are usually pre-determined. For the sake of exposition, it is often convenient to speak in terms of a *binary* relation, which is a subclass over some cartesian product $\mathbf{B}^r \times \mathbf{B}^n$, denoted as $\mathcal{R} \subseteq \mathbf{B}^r \times \mathbf{B}^n$. In general, $\mathbf{B}^r$ and $\mathbf{B}^n$ could themselves be cartesian products of some smaller boolean spaces. Henceforth, unless otherwise noted, we will use relation and binary relation interchangeably.

Definition 2.1 *The* **projection** *of* $\mathbf{x} \in \mathbf{B}^r$ *under relation* $\mathcal{R} \subseteq \mathbf{B}^r \times \mathbf{B}^n$, *is defined as* $\{\mathbf{y} \in \mathbf{B}^n \mid (\mathbf{x}, \mathbf{y}) \in \mathcal{R}\}$ *and is denoted by* $\mathcal{R}(\mathbf{x})$. *Thus* $\mathcal{R}(\mathbf{x}) \subseteq \mathbf{B}^n$ *is a set corresponding to the possible mappings of* $\mathbf{x}$. $\mathcal{R}(\mathbf{x}) = \{\mathbf{y} \in \mathbf{B}^n \mid (\mathbf{x}, \mathbf{y}) \in \mathcal{R}\}$ *is called the* **equivalence class** *of* $\mathbf{x}$.

Definition 2.2 *A relation* $\mathcal{R} \subseteq \mathbf{B}^n \times \mathbf{B}^n$ *is an* **equivalence relation** *on* $\mathbf{B}^n$ *provided* $\mathcal{R}$ *satisfies the following three properties:*

1. **reflexive:** $(\mathbf{x},\mathbf{x}) \in \mathcal{R}, \forall \mathbf{x} \in \mathbf{B}^n$*;*

2. **symmetric:** $(\mathbf{x},\mathbf{y}) \in \mathcal{R} \Rightarrow (\mathbf{y},\mathbf{x}) \in \mathcal{R}$*;*

3. **transitive:** $[(\mathbf{x},\mathbf{y}) \in \mathcal{R}$ and $(\mathbf{y},\mathbf{z}) \in \mathcal{R}] \Rightarrow (\mathbf{x},\mathbf{z}) \in \mathcal{R}$*;*

$\mathbf{x}$ *is* **equivalent** *to* $\mathbf{y}$ *under* $\mathcal{R}$*, written* $\mathbf{x} \sim \mathbf{y}$*, if* $(\mathbf{x},\mathbf{y}) \in \mathcal{R}$*. The equivalence class containing the element* $\mathbf{x}$ *is also denoted as* $[\mathbf{x}] = \{\mathbf{y} \mid (\mathbf{x},\mathbf{y}) \in \mathcal{R}\}$.

An important property of an equivalence relation $\mathcal{R}$ $(\sim)$ on $\mathbf{B}^n$ is that $\mathcal{R}$ induces a partition π on $\mathbf{B}^n$ into disjoint non-vacuous subsets $\pi(\mathbf{B}^n) = \{S_1, S_2, \ldots, S_q\}$, each of which is an equivalence class, such that following are satisfied:

1. $S_i \neq \emptyset$, for each i;

2. $\bigcup_i S_i = \mathbf{B}^n$;

3. $S_i \cap S_j = \emptyset, \forall i \neq j$;

The set $\{S_i \mid 1 \leq i \leq q\}$ is said to be a **partition of** $\mathbf{B}^n$.

2.3 Compatible Functions and Mappings

The relationship between functions and relations can be stated as follows:

Definition 2.3 *A multi-output boolean function* $\mathbf{f} : \mathbf{B}^r \rightarrow \mathbf{B}^n$ *is a* **compatible mapping** *of* $\mathcal{R} \subseteq \mathbf{B}^r \times \mathbf{B}^n$ *if* $\forall \mathbf{x} \in \mathbf{B}^r, \exists \mathbf{y} \in \mathcal{R}(\mathbf{x}) \wedge \mathbf{y} = \mathbf{f}(\mathbf{x})$*. This is denoted by* $\mathbf{f} \prec \mathcal{R}$*.*

Definition 2.4 *Two functions* $\mathbf{f}_1$ *and* $\mathbf{f}_2$ *are* **compatible** *with respect to the boolean relation* $\mathcal{R}$ *if and only if* $\mathbf{f}_1 \prec \mathcal{R} \wedge \mathbf{f}_2 \prec \mathcal{R}$*.*

In general, there are many possible compatible functions corresponding to a relation. In fact, a function is a special case of a relation. A function $\mathbf{f} : \mathbf{B}^r \rightarrow \mathbf{B}^n$ can be written as a relation $\mathcal{F} \subseteq \mathbf{B}^r \times \mathbf{B}^n$ and computed as follows:

$$\mathcal{F} = \prod_{i=1}^{n} (y_i \overline{\oplus} f_i) \tag{1}$$

where $f_1 \ldots f_n$ corresponds to the individual output functions of the multi-output function f. When $f_1 \ldots f_n$ correspond to the next state functions of a finite state machine, then the characteristic function $\mathcal{F}$ is also referred to as the **transition relation.** Similarly, a binary relation $\mathcal{F} \subseteq \mathbf{B}^r \times \mathbf{B}^n$

determines a unique function $\mathbf{f} : \mathbf{B}^r \rightarrow \mathbf{B}^n$ if $\forall \mathbf{x} \in \mathbf{B}^r$, $\#\mathcal{F}(\mathbf{x}) = 1$, meaning $\mathbf{x}$ has a unique projection. Such a relation can be easily converted to a multiple output function as follows:

$$f_i = (\exists \mathbf{y})(\mathcal{F} \cdot y_i) \tag{2}$$

where $\mathbf{y} = \{y_1, \ldots, y_n\}$ are the variables of the range. Here, $\mathcal{F}$ is a characteristic function in terms of $\mathbf{x} = \{x_1, \ldots, x_r\}$ and $\mathbf{y} = \{y_1, \ldots, y_n\}$ variables. Whenever convenient, we shall think of a function as a relation defined by Equation 1. Likewise, a binary relation can be interpreted as a set of functions if the first component of the relation is interpreted as the *domain* and the second component the *range*.

3 Symbolic Class Manipulation

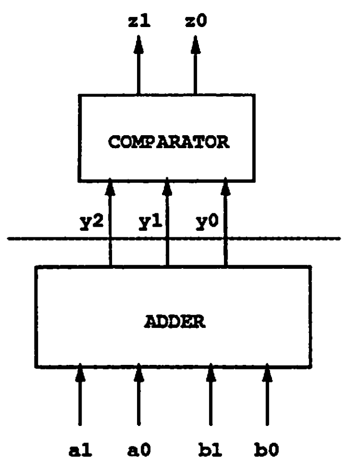
3.1 The Problem

Many problems are fundamentally dependent on the ability to manipulate *equivalence classes* efficiently. To illustrate the problem, we will use the analysis of hierarchically defined boolean networks as an example application where the problem of equivalence classes arises. Consider the simple example shown in figure 1 (borrowed from [2]). It is a cascaded network of an adder followed by a comparator. From the point of view of the comparator, two output patterns of the adder are deemed *equivalent* if they yield the same output at the comparator. For example, the output patterns 001 and 010 are equivalent. The set of equivalent output patterns represents an equivalence class. For this simple example, we can in fact determined that the output patterns of the adder $(y_0 y_1 y_2)$ fall into the following equivalence classes:

$$\begin{array}{rc}
\text{equivalence class 1:} & \{000, 001, 010\} \\
\text{equivalence class 2:} & \{011, 100\} \\
\text{equivalence class 3:} & \{101, 110, 111\}
\end{array}$$

The problem arises when the number of variables becomes too large. In general, the number of equivalence classes along a cut of a network can be exponential in the number of variables in the cut. In practice, the number of variables n is often very large in many problem instances, thus making explicit manipulation techniques prohibitive.

Using BDD's and the concept of characteristic functions, we can easily compute an equivalence relation $E \subseteq \mathbf{B}^n \times \mathbf{B}^n$ corresponding to the set of all equivalent pairs of output patterns. Let



y2y1y0	z1z0
0 0 0	0 0
0 0 1	0 0
0 1 0	0 0
0 1 1	0 1
1 0 0	0 1
1 0 1	1 1
1 1 0	1 1
1 1 1	1 1

Figure 1: Cascaded Adder to Comparator Example.

$A : \mathbf{B}^m \rightarrow \mathbf{B}^n$ be the head subnetwork and let $C : \mathbf{B}^n \rightarrow \mathbf{B}^r$ be the tail subnetwork. The equivalence relation E at the output of subnetwork A can be implicitly computed as follows:

$$E(\mathbf{u},\mathbf{v}) = (\exists \mathbf{y})(C(\mathbf{u},\mathbf{y}) \cdot C(\mathbf{v},\mathbf{y})).$$

This relation E is referred to as the *cross-observability* relation [5]. Continuing with the example shown in figure 1, the relation E is a characteristic function representing the following equivalent pairs:

{000, 000}	{000, 001}	{000, 010}
{001, 000}	{001, 001}	{001, 010}
{010, 000}	{010, 001}	{010, 010}
{011, 011}	{011, 100}	
{100, 011}	{100, 100}	
{101, 101}	{101, 110}	{101, 111}
{110, 101}	{110, 110}	{110, 111}
{111, 101}	{111, 110}	{111, 111}

While basic machinery exists for computing this pairwise relation (*viz.* BDD's and characteristic functions), we do not have at our disposal analogous machinery for computing and representing the *equivalence classes* efficiently (other than straightforward truth table enumeration). Therefore, we need to develop the missing machinery for symbolically manipulating equivalence classes. To manipulate equivalence classes symbolically, we implicitly derive an *encoding function* that encodes each equivalence class to a unique member in the equivalence class. Consider the following problem to be solved:

Problem 3.1 : Let $\mathcal{R} \subseteq D \times \Sigma$ be a one-to-many relation, where $D \subseteq \mathbf{B}^r$ and $\Sigma \subseteq \mathbf{B}^n$. Compute a compatible function $\mathcal{F} \subseteq D \times \Sigma$ such that the following properties are satisfied:

1. $\forall \mathbf{x} \in D$, $\#\mathcal{F}(\mathbf{x}) = 1$ and $\mathcal{F}(\mathbf{x}) \in \mathcal{R}(\mathbf{x})$;

2. $\mathcal{R}(\mathbf{u}) \equiv \mathcal{R}(\mathbf{v})$ implies $\mathcal{F}(\mathbf{u}) = \mathcal{F}(\mathbf{v})$.

The second property states that if $\mathbf{u}$ and $\mathbf{v}$ in D have the same equivalence class, then the encoding function generated should produce the same output mapping.

Continuing with the above example, a solution to problem 3.1 would produce the following encoding function:

$u_2u_1u_0$	$v_2v_1v_0$
{000, 001, 010}	{000, 001, 010}
{011, 100}	{011, 100}
{101, 110, 111}	{101, 110, 111}

$\Rightarrow$

$u_2u_1u_0$	$v_2v_1v_0$
{000, 001, 010}	000
{011, 100}	011
{101, 110, 111}	101

Note that each element in the range of the encoding function uniquely corresponds to an equivalence class of the equivalence relation. To derive the above encoding function, we propose a new BDD boolean operator called the *compatible projection* operator. This is described next.

3.2 The Compatible Projection Operator

To derive a compatible function that uniquely encodes each equivalence class, we choose a criterion that uniquely orders the elements in the co-domain (output space) such that among the output choices for each $\mathbf{x} \in \Sigma$ (i.e. $\mathcal{R}(\mathbf{x})$), the element lowest in the order is selected. This can be performed by using a *distance* metric to determine a total ordering on the choices.

Definition 3.1 *Let $\mathbf{B}^n$ be a n-dimensional boolean space and $y_1 \prec y_2 \prec \ldots \prec y_n$ be an ordering of its variables. The distance between two vertices $\alpha \in \mathbf{B}^n$ and $\beta \in \mathbf{B}^n$ is as defined in [6, 14]:*

$$d(\alpha, \beta) = \sum_{i}^{n} | \alpha_i - \beta_i | \, 2^{n-i}$$

Using the distance operator, we can define an ordering on the vertices of a boolean space relative to some reference vertex.

Lemma 3.1 *Given a variable ordering $y_1 \prec y_2 \prec \ldots \prec y_n$, and a reference vertex $\alpha \in \mathbf{B}^n$, the distance from α to each vertex $\sigma \in \mathbf{B}^n$ corresponds to a non-negative injective integer mapping $d : \mathbf{B}^n \rightarrow \mathbf{Z}^+$.*

Proof: It follows from definition that $d(\alpha, \sigma)$ is a non-negative integer $\forall \sigma \in \mathbf{B}^n$. Now suppose the mapping is not injective. It implies that there exists $\sigma_1 \in \mathbf{B}^n$ and $\sigma_2 \in \mathbf{B}^n$, $\sigma_1 \neq \sigma_2$, such that $d(\alpha, \sigma_1) = d(\alpha, \sigma_2)$. This is a contradiction. □

Lemma 3.2 *Let $\mathbf{B}^n$ be a n-dimensional boolean space. Given a variable ordering $y_1 \prec y_2 \prec \ldots \prec y_n$, and a reference vertex $\alpha \in \mathbf{B}^n$, the distance operator $d(\alpha, \sigma)$ induces a unique ordering on the vertices in $\mathbf{B}^n$, denoted as $\sigma_1 \prec \sigma_2 \prec \ldots \prec \sigma_{2^n}$.*

Proof: The ordering of $\sigma_i \in \mathbf{B}^n$ corresponds exactly to the ordering of integer values mapped from d. □

From the foregoing lemmas, we can uniquely determine an ordering on the vertices of an arbitrary size boolean space based solely on the variable ordering and a reference vertex in the boolean space. The reference vertex can be any point in $\mathbf{B}^n$. Depending on the application, this reference vertex is often naturally determined. For example, in state minimization (c.f. section 4), the reset state of the machine is used. Another useful reference vertex is the all 0's vector. This result can be generalized to arbitrary sets of boolean vectors.

Theorem 3.3 *Let $\Sigma \subseteq \mathbf{B}^n$ be a subset of a n-dimensional boolean space. Given a variable ordering $y_1 \prec y_2 \prec \ldots \prec y_n$, and a reference vertex $\alpha \in \mathbf{B}^n$, the distance operator $d(\alpha, \sigma)$ induces a unique total ordering on the vertices in Σ.*

Proof: Follows from lemmas 3.1 and 3.2. □

Note that the reference vertex α need not be in the set Σ. This permits us to define a consistent ordering on any subsets. Using the same reference vertex, the *relative* ordering between subsets is also well defined. Using the distance metric, we can now defined a selection operator that can be used to select a unique member from a set.

Definition 3.2 *Define*

$$\perp(\alpha, \Sigma) = \arg\min_{\sigma \in \Sigma} d(\alpha, \sigma)$$

to be the **closest interpretation** *of α in Σ.*

Example 3.1 : Suppose $\Sigma = \{111, 010, 101\}$. Let $\alpha = 000$ be the reference vertex. The corresponding distances are $d(000, 111) = 7$, $d(000, 010) = 2$, and $d(000, 101) = 5$, which corresponds to the ordering $010 \prec 101 \prec 111$. The closest interpretation is $\perp(\alpha, \Sigma) = 010$. If the reference vertex is instead $\alpha = 110$. Then the ordering is $111 \prec 101 \prec 010$ and the closest interpretation is $\perp(\alpha, \Sigma) = 111$.

We can now define the compatible projection operator.

Definition 3.3 *Let $\mathcal{R} \subseteq D \times \Sigma$ be a binary relation, where $D \subseteq \mathbf{B}^r$ and $\Sigma \subseteq \mathbf{B}^n$. Let $\alpha \in B^n$ a reference vertex. The* **compatible projection,** *or simply* **c-projection,** *of $\mathcal{R}$ relative to α, denoted as* **projection**$(\mathcal{R}, \alpha)$, *is the compatible function $\mathcal{F}$ defined as follows:*

$$\mathcal{F} = \{(\mathbf{x}, \mathbf{y}) \mid (\mathbf{x}, \mathbf{y}) \in \mathcal{R} \ \text{ and } \ \mathbf{y} = \perp(\alpha, \mathcal{R}(\mathbf{x}))\}$$

The compatible projection operation can be computed very efficiently with BDD's. The characteristic function $\mathcal{F}$ can be converted to multi-output form using equation 2. For greater generality, we treat $\mathcal{R} \subseteq \mathbf{B}^r \times \mathbf{B}^n$ as a characteristic function over the cartesian space $\mathbf{B}^r \times \mathbf{B}^n$. We let α be a cube in $\mathbf{B}^r \times \mathbf{B}^n$ (i.e. $\alpha = \alpha_1 \alpha_2 \ldots \alpha_{r+n}$) where $\alpha_i = \{0, 1, *\}$. If $\alpha_i = *$, then the corresponding variable x_i is considered part of the domain, and the range otherwise. The set of literals $\alpha_i \neq *$ defines a reference vertex in the range. In this case, α cannot be NULL. The algorithm is given in Figure 2.

Lemma 3.4 *If $\alpha = 1$, then $\mathcal{R} =$* **projection**$(\mathcal{R}, \alpha)$. *If $\mathcal{R} = 1$, then $\mathcal{F} = \mathcal{R} \cdot \alpha$.*

These represent interesting special cases that can be easily computed. We now give some theoretical results pertaining to this operator.

3.3 Main Results

The compatible projection operator indeed implements a solution to problem 3.1.

Lemma 3.5 *The compatible projection operator satisfies all the properties outlined in problem 3.1.*

```
function projection($r$,$\alpha$) {
    if ($\alpha$ = NULL) return error;
    if ($\alpha = 1$ or $r = 0$) return $r$;
    if ($r = 1$) return $r \cdot \alpha$;
    let $\alpha_i$ = literal_of_topvar($\alpha$);
    let $y$ = support($\alpha$);
    if ($\exists y.r_{\alpha_i} = 1$) return $\alpha_i \cdot$ projection($r_{\alpha_i}, \alpha_{\alpha_i}$);
    else if ($\exists y.r_{\alpha_i} = 0$) return $\overline{\alpha_i} \cdot$ projection($r_{\overline{\alpha_i}}, \alpha_{\alpha_i}$);
    else return $\alpha_i \cdot$ projection($r_{\alpha_i}, \alpha_{\alpha_i}$)
                    + $\overline{\alpha_i} \cdot$ projection($r_{\overline{\alpha_i}} - \exists y.r_{\alpha_i}, \alpha_{\alpha_i}$);
}
```

Figure 2: The Compatible Projection Algorithm.

Proof: The $\perp(\alpha, \mathcal{R}(\mathbf{x}))$ operator by definition selects exactly one **y** mapping from $\mathcal{R}(\mathbf{x})$ for every **x**. This satisfies the first condition. The $\perp$ operator also guarantees that the mapping with the lowest cost, according to the distance metric relative to α, will be chosen. This implies that if **u** and **v** have the same output choices, the lowest cost choice is both cases will be the same. □

This in fact leads to the following fundamental theorem.

Theorem 3.6 (Canonical Compatible Mapping Theorem) *Given $\mathcal{R} \subseteq \mathbf{B}^r \times \mathbf{B}^n$, $\alpha \in B^n$, and a variable ordering $y_1 \prec y_2 \prec \ldots \prec y_n$ on B^n, the compatible mapping $\mathcal{F} =$* **projection**$(\mathcal{R}, \alpha)$ *is* **canonical.**

Proof: Follows from the above arguments. □

This result is significant since it tells us that a unique mapping can always be derived. Moreover, if $\mathcal{R}$ is an equivalence relation, we can show that the new function $\mathcal{F}$ indeed encodes all members of the same equivalence class to a unique member in the equivalence class.

Theorem 3.7 (Equivalence Class Theorem) *Let $\mathcal{R} \subseteq \mathbf{B}^n \times \mathbf{B}^n$ be an equivalence relation on $\mathbf{B}^n$. Let $\pi(\mathbf{B}^n) = \{S_1, S_2, \ldots, S_q\}$ be the partition of disjoint equivalence classes induced by $\mathcal{R}$, and* *let* $\mathcal{F} = \textbf{projection}(\mathcal{R}, \alpha)$ *be the compatible projection of* $\mathcal{R}$. *Then*

$$\forall S_i \in \pi(\mathbf{B}^n), \forall \mathbf{u}, \mathbf{v} \in S_i, \mathcal{F}(\mathbf{u}) = \mathcal{F}(\mathbf{v})$$

Proof Sketch: This is due to the fact that if $\mathbf{u}$ and $\mathbf{v}$ belong to the same equivalence class in $\mathbf{B}^n$, then they will exactly the same set of possible mappings. Since the lowest cost mapping will be selected in both cases, they will have the mapping in $\mathcal{F}$. $\square$

The significance of this result is that the function produced by the compatible projection operator uniquely encodes each equivalence class.

4 An Example Application

We illustrate in this section the technique of symbolic class manipulation on the problem of exact state minimization. In practice, large sequential circuits are generally not state minimal. This is especially true when the sequential circuits are automatically compiled from high-level descriptions. Another common source of non-minimal state machines arises from synthesizing interacting networks of finite state machines. It is well know that these finite state machines typically contain a large number of equivalent states. When considered as a whole, the cartesian product of the component state spaces typically result in state explosion. One approach to simplifying the machine is to perform state minimization. There are well known techniques and algorithm for solving this problem when a state transition graph model can be extracted [9]. The complexity of this algorithm is $O(Q \log_2 Q)$ where Q is the number of states. However, Q is exponential in the number of state registers in the worst case. Hence, known algorithms such as those in [9] are not applicable for a large class of problems. For example, in the machine *mkey* (see section 5), obtained from the data encryption standard chip (DES) [15], there are over 10^{68} states. We emphasize that the true complexity of the state space strongly depends on the actual structure of the sequential circuit rather than the number of registers.

Using the machinery developed in the previous section, we have developed an exact algorithm that can feasibly state minimize finite state machines of such sizes. For synthesis, a state minimized machine will often lead to a more efficient realization and fewer state registers to implement, which would also ease the testing problem. For verification, the ability to implicitly compute a reduced machine is also extremely important since design properties can often be checked on the reduced machine, but potentially much more efficiently since the state space may be drastically reduced.

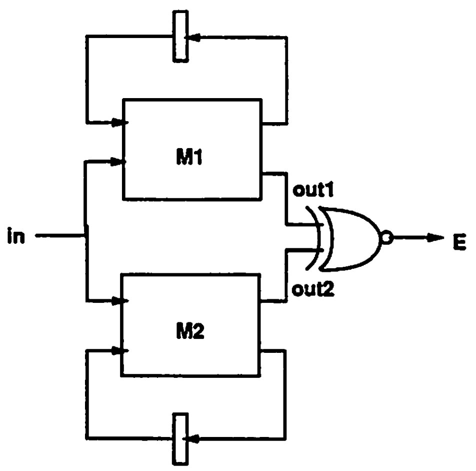


Figure 3: Product Machine for Equivalence Analysis.

Also, there may be specific verification queries that can only be checked or are more naturally checked on the reduced machine. The state minimization technique can also be used to simplify the BDD representation of the transition relation. Before describing the procedure, we first give some basic working definitions.

A finite state machine M is a 6-tuple $\langle \Sigma, Q, \mathcal{O}, \delta, \lambda, q_0 \rangle$, where Σ is the input alphabet, Q is a finite set of states, $\mathcal{O}$ is the output alphabet, and $q_0 \in Q$ is the initial state of the machine (in general, the initial condition may be a set). An alphabet is a set of symbols. For the sake of simplicity, we will assume an alphabet is defined over a finite n-dimensional boolean space $\{0,1\}^n$ where each vertex (minterm) uniquely represents a symbol in the alphabet: eg. $Q = \{0,1\}^n$, $\Sigma = \{0,1\}^r$, and $\mathcal{O} = \{0,1\}^m$. As explained earlier, non-boolean domains can be easily handled by applying an encoding function $\xi : D \rightarrow \mathbf{B}^k$ which maps one-to-one each element of D to a unique boolean vector in $\{0,1\}^k$.

The outline of the exact state minimization algorithm is as follows:

1. Given a finite state machine M, first compute the equivalence relation $E \subseteq \mathbf{B}^n \times \mathbf{B}^n$ corresponding to the set of all equivalent state pairs. Two states q_i and q_j are equivalent if and only if $(q_i, q_j) \in E$. The relation E is computed by first building a product machine $M^* = M_1 \otimes M_2$, as shown in figure 3. The set of equivalent state pairs $E \subseteq \mathbf{B}^n \times \mathbf{B}^n$ can then be efficiently computed using the BDD-based symbolic traversal algorithms described in [11].

2. The relation E is a one-to-many mapping that maps each state $q \in \mathbf{B}^n$ to any one of its

equivalent state $[q]$. We use the compatible projection operator to find a compatible function $S \subseteq \mathbf{B}^n \times \mathbf{B}^n$ that maps all equivalent states to the same state (the function S can be converted to a function form $S : \mathbf{B}^n \rightarrow \mathbf{B}^n$ viz equation 2). The function S is effectively an encoding function that re-encodes all equivalent state codes to a unique code.

3. Modify the transition relation using the function S.

In the first step, the relation E is an equivalence relation represented in BDD form. Depending on the example, there may be a very large number of equivalent state pairs. Referring to again the example *mkey*, there are over 10^{130} equivalent state pairs. The algorithms described in [11] were able to feasibly compute all equivalent state pairs with reasonable CPU time. The efficiency is dependent on the regularity of the underlying structure. Fortunately, a large number of circuits belong to this class. We refer the interested reader to [11] for the details on how the equivalent state pairs are computed.

In the second step, the function S can be computed as follows:

$$S = \mathbf{projection}(E, q_0) \tag{3}$$

By default, the reset state q_0 is used as the reference vertex. This guarantees the reset state code is retained (i.e. all states equivalent to the reset state will be re-assigned the reset state code). However, this is not a necessary condition. The reduced set of states can be computed as follows:

$$Q = \text{set of reduced states} = (\exists \mathbf{u}) S \tag{4}$$

where $\mathbf{u}$ denotes the set of variables corresponding to the first component of S. This corresponds effectively to the *range* of S. The number of elements in the set Q corresponds to the number of reduced states:

$$\#Q = \text{number of reduced states} \tag{5}$$

It also corresponds exactly to the number of equivalence classes in E. Using S, the transition relation of the finite state machine can be simplified as follows:

Definition 4.1 *The next state transition relation for a finite state machine M is $T \subseteq \mathbf{B}^r \times \mathbf{B}^n \times \mathbf{B}^n$ such that $(i, \mathbf{x}, \mathbf{y}) \in T$ if and only if the state $\mathbf{y}$ can be reached in exactly one state transition from state $\mathbf{x}$ when input i is applied.*

The transition relation implicitly defines the finite state machine. Here, i denotes the primary input variables, x the present state variables, and y the next state variables. We can then compute the state minimized transition relation as follows:

$$T_{min}(i, \mathrm{u}, \mathrm{v}) = (\exists \mathrm{y})[(\exists \mathrm{x})[T(i, \mathrm{x}, \mathrm{y}) \cdot S(\mathrm{x}, \mathrm{u})] \cdot S(\mathrm{y}, \mathrm{v})]$$

In the above equation, we explicitly stated the variable names in the parentheses for each relation to illustrate the correspondence. The state minimized transition relation T_{min} can be re-expressed in terms of the x and y variables by variable substitution.

5 Preliminary Results

In this section, we present some preliminary results on the concept of symbolic class manipulation described in this paper. We demonstrate the capabilities of the proposed techniques by applying the techniques to the problem of state minimization for large sequential circuits. The algorithm makes use of the the compatible projection operator described in this paper, which we have efficiently implemented in BDD's. All experimental results presented were measured on a IBM RS6000 workstation and the CPU times presented are quoted in seconds.

The benchmarks used were obtained from various industrial and university sources in the form of a gate-level description. The experiments were designed to show the limitation of conventional explicit manipulation methods and expose the need for *symbolic* manipulation techniques. The examples `s208` and `s298` were obtained from the ISCAS sequential benchmark set. The example `tlc` corresponds to a traffic light controller. The examples `vit3` and `viterbi` are control circuits obtained from the VITERBI speech recognition processor chip [13]. The example `key` is derived from a control circuit that implements the key encryption algorithm in the data encryption standard (DES) chip [15]. It contains a large number of data registers. The examples `mkey` and `tkey` were derived from `key` by considering a subset of outputs (namely the control outputs). These examples vary in complexity with the largest one having 228 latches and over 10^{68} states. Since BDD's are used, number of registers is no longer the bottleneck. We emphasize that the "real" complexity of the circuit is actually dependent on the structure and regularity of the problem.

Our experiment for symbolic class manipulation, using state minimization as a test application, was done as follows. We begin with each benchmark sequential circuit starting at the gate netlist level. For each example, we first computed the set of all equivalent state pairs using the BDD-based computation algorithms described in [11]. The algorithms described in [11] are guaranteed

circuit	i/o/lits(fac)	set size	eq. pairs	eq. classes	regs.	lower	CPU1	CPU2
s208	11/21/166	256	3310	40	8	6	0.45	0.19
s298	3/6/244	16384	510000	8060	14	13	35.88	15.83
tlc	3/5/324	1020	25400	254	10	8	5.41	1.43
vit3	11/4/880	512	18400	15	9	4	5.08	0.13
viterbi	11/34/1372	4100	10700	3120	12	12	17.50	0.48
mkey	258/10/3676	4.31e+68	1.11e+130	1.68e+07	228	24	146.16	17.21
tkey	258/20/3686	4.31e+68	1.06e+124	1.76e+13	228	44	177.83	21.87
key	258/193/3865	4.31e+68	4.31e+68	4.31e+68	228	228	517.85	24.67

Table 1: State Minimization Results

i/o/lits(fac):	number of primary inputs and outputs, and literals in factored form
set size:	number of states in the initial machine
eq. pairs:	number of equivalent state pairs
eq. classes:	number of equivalence classes and states after state minimization
regs.:	initial number of registers
lower:	lower bound on minimum code length for re-encoding reduced machine
CPU1:	CPU time for computing the set of equivalent state pairs
CPU2:	CPU time for performing state minimization
	CPU times reported in seconds on an IBM RS6000 Workstation

to find all possible equivalent state pairs. These algorithms were implemented using Berkeley's BDD package [1] and implicit enumeration package [14]. The set of equivalent state pairs, which corresponds to an equivalence relation, is represented as a characteristic function in BDD form.

An encoding function that encodes the equivalent states is then derived implicitly using the compatible projection operator, as described in sections 3.2 and 4. The transition functions of the reduced machine were accordingly constructed. The number of reduced states after state minimization is usually much less than the number of states in the initial form. Thus, the reduced states can potentially be re-encoded with significantly fewer number of state registers.

Table 1 shows the result of the above experiment. Some basic statistics for each benchmark circuit is shown in the second column. The total number of states for each example is indicated under the column labeled **set size**. We quote here the sum of both reachable and unreachable states. This is because we consider both reachable and unreachable states when deciding state equivalence. This is more general, but equivalent states analysis for the reachable subset is a trivial extension. The largest examples are `mkey`, `tkey`, and `key`, each of which has 4.31×10^{68} possible states. The number of equivalent state pairs computed for each example is indicated under the column labeled **eq. pairs**. For the example `mkey`, there were over 10^{130} equivalent state pairs.

Using the compatible projection operator, we were able to implicitly compute the number of equivalence classes corresponding to the equivalence relation and merge all equivalent states in the finite state machine. The column labeled **eq. classes** indicates the number of equivalence classes for each finite state machine example. This is also the number of states after state minimization. For example, the machine `mkey` was state minimized from 4.31×10^{68} states to only 1.68×10^{7} states. It should be noted that the number of equivalence classes can be in general be the same as number of states (i.e. no equivalent states), which is exponential in the number of state variables. Hence, explicit counting of equivalence classes is usually not possible. The relationship between the number of equivalence classes and the number of states is dependent on the specific example.

Because the number of state patterns after state minimization may be considerably less, it is possible to encode the reduced states with fewer number of state registers. We give in the column labeled **regs.** the number of state registers for each example before state minimization. The lower bounds on the number of registers to re-encode the reduced state spaces are given under the column labeled **lower**. For example, the minimum code length to re-encode the reduced machine `tkey` is 44 registers, but the original register count was 228.

The CPU times for computing the equivalence relation using the algorithms described in [11, 14] are indicated in the column labeled **CPU1**, and the CPU times for computing the equivalence classes (using compatible projection) and performing state reduction are indicated in the last column labeled **CPU2**. In all cases, the CPU time required for performing symbolic class manipulation is relatively modest. In fact, the CPU time for computing the equivalence relation **strictly dominates** the overall CPU time of the state minimization process.

6 Conclusion

In this paper, we have provided new core machinery for symbolically manipulating equivalence relations and classes efficiently. We have shown the relevance of this fundamental concept by giving examples of applications. We have described the compatible projection operator as a means for finding a compatible function corresponding to a given relation. A fundamental property of the compatible projection operator is that the function produced is canonical. In manipulating equivalence classes, the compatible projection operation implicitly derives an encoding function that encodes the equivalence class information symbolically. Experimental results demonstrate the practical importance of the proposed methods. Since we "encode" the equivalence classes with only $\log_2 N$ number of variables, where N is the number of equivalence classes, the BDD function that represents the encoding of the equivalence classes is often very compact. Hence, very large problem sizes can be handled. The main limitation is the size of the BDD representing the encoding function. Empirically, we have not found this to be a bottleneck on the problems tested.

References

[1] K.L. Brace, R.L. Rudell, and R.E. Bryant. Efficient implementation of a bdd package. In *Design Automation Conference*, June 1990.

[2] R.K. Brayton and F. Somenzi. Minimization of boolean relations. In *International Conference on Computer-Aided Design*, November 1989.

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

[4] J.R. Burch, E.M. Clarke, K.L. McMillan, and D.L Dill. Sequential circuit verification using symbolic model checking. In *Design Automation Conference*, June 1990.

[5] E. Cerny and C. Mauras. Tautology checking using cross-controllability and cross-observability relations. In *International Conference on Computer-Aided Design*, November 1990.

[6] O. Coudert, C. Berthet, and J.C. Madre. Verification of sequential machines using function vectors. In L.J.M. Claesen, editor, *Formal VLSI Correctness Verification*. Elsevier Science Publishers B.V., North Holland Press, 1990.

[7] T. Hwang, R.M. Owens, and M.J. Irwin. Exploiting communication complexity for multi-level logic synthesis. *IEEE Transactions on Computer-Aided Design*, 9(10):1017–1027, October 1990.

[8] R.M. Karp. Function decomposition and switching circuit design. *Journal of Society of Industrial Applied Mathematics*, 11(2), June 1963.

[9] Z. Kohavi. *Switching and Finite Automata Theory*. McGraw Hill, 1978.

[10] B. Lin and F. Somenzi. Minimization of symbolic relations. In *International Conference on Computer-Aided Design*, November 1990.

[11] B. Lin, H. J. Touati, and A. R. Newton. Don't care minimization of multi-level sequential logic networks. In *International Conference on Computer-Aided Design*, November 1990.

[12] S. Muroga, Y. Kambayashi, H. C. Lai, and J. N. Culliney. The transduction method: Design of logic networks based on permissible functions. *IEEE Transactions on Computer-Aided Design*, 10(10):1404–1424, October 1989.

[13] A. Stolzle. A VLSI wordprocessing subsystem for a real time large vocabulary continuois speech recognition system. In *MS Thesis*, September 1989.

[14] H. J. Touati, H. Savoj, B. Lin, R. K. Brayton, and A. Sangiovanni-Vincentelli. Implicit state enumeration of finite state machines using bdd's. In *International Conference on Computer-Aided Design*, November 1990.

[15] National Bureau of Standards U.S. Department of Commerce. Data encryption standard. In *Federal Information Processing Standards Publication (FIPS PUB 46)*, January 1977.