An $n \log n$ Algorithm for Online BDD Refinement*

Nils Klarlund

AT&T Labs Research
600 Mountain Ave.
Murray Hill, NJ 07974
klarlund@research.att.com

Abstract. Binary Decision Diagrams are in widespread use in verification systems for the canonical representation of finite functions. Here we consider multi-valued BDDs, which represent functions of the form $\varphi : \mathbb{B}^\nu \to \mathcal{L}$, where $\mathcal{L}$ is a finite set of leaves.
We study a rather natural online BDD refinement problem: a partition of the leaves of several shared BDDs is gradually refined, and the equivalence of the BDDs under the current partition must be maintained in a discriminator table. We show that it can be solved in $O(n \log n)$ if n bounds both the size of the BDDs and the total size of update operations. Our algorithm is based on an understanding of BDDs as the fixed points of an operator that in each step splits and gathers nodes.
We apply our algorithm to show that automata with BDD-represented transition functions can be minimized in time $O(n \cdot \log n)$, where n is the total number of BDD nodes representing the automaton. This result is not an instance of Hopcroft's classical algorithm for automaton minimization, which breaks down for BDDs because of their path compression property.

1 Introduction

Binary Decision Diagrams [3] form the backbone of many symbolic methods for verification of hardware and software. BDDs are essentially acyclic automata whose state spaces are shrunk by a technique called *path compression.* More precisely, a BDD is an acyclic, rooted, directed graph that represents a function $\varphi : \mathbb{B}^\nu \to \mathcal{L}$ from ν Boolean variables to a finite codomain $\mathcal{L}$ of leaves. (These BDDs are sometimes called Multi-Terminal BDDs to distinguish them from two-terminal BDDs that denote Boolean functions.)

The Problem

We consider the following problem, which is at the heart of the minimization problem for BDD-represented automata. We are given several functions

* This work was carried out while the author was with BRICS, Department of Computer Science, Aarhus, Denmark.

$\phi_i : \mathbb{B}^{V} \to \mathcal{L}$. They can be represented by a shared BDD, which is an acyclic, directed graph with a distinguished root for each i, where each root induces a subgraph that constitutes a BDD for ϕ_i. Now given a partition of the leaves (or codomain), we would like to calculate a *function discriminator*, which associates a discriminator value $R(i)$ to each i such that $R(i) = R(j)$ if and only ϕ_i and ϕ_j are equivalent under the leaf partition, i.e., if for all $\mathbf{u} \in \mathbb{B}$, $\phi_i(\mathbf{u})$ is equivalent to $\phi_j(\mathbf{u})$. Note that the leaf partition itself can also be represented by a discriminator D such that v and v' are equivalent if and only if $D(v) = D(v')$.

The online version of this problem is to maintain the function discriminator after an online operation specifies an *update*, which is a further refinement of the current leaf partition. Initially, the leaf partition consists of only one equivalence class, i.e., all functions ϕ_i are equivalent and D and R are a constant functions.

A simple algorithm for the online BDD refinement problem can be based on the linear time reduction of BDDs [12]: after each refinement operation, the whole BDD structure can be reduced to a canonical BDD for the functions that map into equivalence classes. This strategy implies that each node is touched potentially as many times as the number of operations. Thus an $O(n^2)$ algorithm arises, if we assume n online operations.

Our Solution

In this paper, we formulate a more efficient algorithm, which runs in time $O(n \min(k, \log n) + k)$, where n is the number of nodes in the BDDs and k is the total size of all update operations. Thus, if n also bounds k, then the algorithm is $O(n \log n)$.

Unfortunately, no simple solution seems to achieve $O(n \log n)$. Instead, our analysis proceeds roughly as follows.

For BDDs, the *Split* operation of partition refinement algorithms such as [11] does not directly yield a partition refining the current one. Rather, the result of a split operation, which we call a *decision partition* must be followed by a *Grow* operation that gathers all nodes equivalent under path compression. We show that the canonical BDD representation of ϕ can be obtained as the fixed point of *Grow* $\circ$ *Split* (even though this composed operator is not monotone). This characterization is not surprising, since usual BDD algorithms are also able to calculate a canonical representation in one sweep.

The *Grow* operation cannot be used with Hopcroft's "process the lesser half" strategy [8], since all decision blocks must be grown as opposed to the situation in traditional partition refinement algorithms, where the largest blocks created can be ignored.

Fortunately, the canonical BDD can be calculated under weaker assumptions about the fixed point operator. The *Grow* operation can be weakened to an operation, which we call *CGrow* since it allows certain blocks resulting from the normal *Grow* operation to be coalesced. As a result, information is lost. Curiously, it turns out that if a partition is a fixed point under *Split* and *CGrow*, then it is also a fixed point under *Split* and *Grow*.

We use this property in our online algorithm to discard any large block that arises during the iteration of the fixed point operator. The block is discarded by being coalesced with another, smaller block, while the expense of calculating it can be attributed to a third block known also to be small.

Consequences and Comparison to Previous Work

It has been known for a long time [8] that deterministic finite-state automata can be minimized in time $O(m \cdot n \log n)$, where n is the number of states and m is the size of the input alphabet. A recent variation on the standard method yields a similar bound [2].

BDDs allow automata with n states and 2^n letters—each inducing a different behavior in the automaton—to be represented by graphs of polynomial size in n; see [5, 7], where also $O(n^2)$ minimization algorithms are presented. The automaton representation in [5] allows symbolic calculations involving inductive definitions of sequential circuits, whereas the representation in [7] is the backbone of a practical implementation of Monadic Second-order Logic on Strings. For a comparison of these related representations, see [1].

The $O(n^2)$ minimization algorithms are a potential bottleneck for the use of BDD represented automata. In the Mona project at Aarhus (`http://www.brics.dk/~klarlund/MonaFido`), we have observed that for big automata (with thousands of states), the time to minimize using the straightforward algorithm is an order of magnitude larger than the time spent in constructing the automata.

In this paper, we show that our online BDD refinement algorithm allows minimization to be carried out in only $O(n \cdot \log n)$ steps, where n is the size of the representation. To our knowledge, the only other algorithm for large alphabets that reach a similar bound is that of [4], where incompletely specified transition functions are considered. The compression possible with the BDD representation is exponentially greater.

It should also be noted that when automata are represented with BDDs that are not path compressed an $O(n \log n)$ algorithm follows easily by considering the automaton as working on words over $\mathbb{B}$. Path compression, however, seems to be of major practical significance although the asymptotic gain is only slight [10].

Finally, we mention that online minimization of automata on large, implicitly represented state spaces (not alphabets) have been considered in [9]. Online minimization here refers to incremental exploration of the state space. This algorithm bears a superficial resemblance to ours in that it also alternates between minimal and maximal fixed point iterations.

Overview

In Section 2, we define the online BDD refinement problem. We develop a theoretical framework for understanding BDDs as fixed points in Section 3. We show that a weak composed operator suffices for generating the minimum fixed point. In Section 4, we provide a description of our online algorithm, which is based on the weak operator. Section 5 discusses the application of our algorithm to automaton minimization.

2 Online BDD Refinement

Notation

Assume we are given a set $x_0, x_1, \ldots, x_{\nu-1}$ of Boolean variables. A *truth assignment* to these variables is a vector $\mathbf{u} \in \mathbb{B}^\nu$. An *assignment prefix* $\mathbf{u}$ up to i is a truth assignment to variables $x_0, \ldots x_i$. A *Binary Decision Diagram* or *BDD* φ is a rooted, directed graph with the following properties. The root is named $\wedge\varphi$. Each node v in φ is either an *internal node* or a *leaf*. An internal node possesses an *index* denoted $v.i$. Also, it contains edges $v \cdot 0$, which points to a node called the *low successor* of v, and $v \cdot 1$, which points to the *high successor*. The index of a successor of v is always greater than the index of v. A leaf has no successors and no index. Let the set of leaves be $\mathcal{L}$. The graph φ denotes a function, also called φ, from $\mathbb{B}^\nu \to \mathcal{L}$. To calculate $\varphi(\mathbf{x})$, one starts at the root. If the root is a leaf, then the value $\varphi(\mathbf{x})$ is the root; otherwise, let i be the index of the root. If x_i is 1 then go to the high successor, and if x_i is 0 go to the low successor. Continue in this way until a leaf is reached. This leaf is the value of $\varphi(\mathbf{x})$. (Since there may be jumps greater than one in the index of some of the variables, some of the values in the assignment may be irrelevant.) In general, if v is a node of index i and $\mathbf{u}$ is a value assignment to $x_i, \ldots, x_j$, then $v \cdot \mathbf{u}$ denotes the node reached by following $\mathbf{u}$ from v.

The BDD φ defines a partition $\equiv_\varphi$ of assignment prefixes given by $\mathbf{u} \equiv_\varphi \mathbf{u}'$ if $\wedge\varphi \cdot \mathbf{u} = \wedge\varphi \cdot \mathbf{u}'$.

We shall consider the case where the leaves are used to differentiate between finer and finer partitions of $\mathbb{B}^\nu$. The partition is given by a *leaf discriminator* $D : \mathcal{L} \to \mathbb{N}$. Two assignments $\mathbf{x}$ and $\mathbf{y}$ are then equivalent if $D \circ \varphi(\mathbf{x}) = D \circ \varphi(\mathbf{y})$.

BDDs may also be shared. For example, we use $\varphi = \varphi_0, \cdots, \varphi_{n-1}$ to denote a directed graph with roots $\wedge\varphi_i$ such that the nodes reachable from each root constitute a BDD. If D is a discriminator for the leaves, then we say that $R : [n] \to \mathbb{N}$ is a *function discriminator* for $D \circ \boldsymbol{\varphi}$ if $D \circ \varphi_i = D \circ \varphi_j$ iff $R(i) = R(j)$.

Note that if D is a constant discriminator (i.e. if D is a constant function), then all $D \circ \varphi_i$ are equivalent.

The Problem

The BDD online refinement problem is to maintain a function discriminator R for $D \circ \boldsymbol{\varphi}$ when D is updated piecemeal. Each update operation specifies a partial mapping $E : \mathcal{L} \to \mathbb{N}$, which defines the change to D. Thus, if

$$D'(v) = \begin{cases} D(v) \text{ if } v \notin \mathbf{domain}(E) \\ E(v) \text{ if } v \in \mathbf{domain}(E) \end{cases}$$

then the new value of D is D'. In order for the new D to specify a partition refining the one given by the current D, we require that the range of E is disjoint from the range of the current D. The time requirements of our algorithm will prevent it from updating all values $R(i)$ with each iteration. Thus, we require as an additional output after each update operation the list of i for which $R(i)$ has changed. The desired functionality can be summarized as follows.

BDD Online Refinement Problem
Input: n shared BDDs φ with leaves $\mathcal{L}$ and discriminator D, which is initially a constant function.
Maintained : A functional discriminator R of length n.
Update: A partial mapping $E : \mathcal{L} \to \mathbb{N}$ such that **range**(E) does not intersect the current leaf discriminator. The leaf discriminator D is updated according to E as explained above. After each update operation, the contents of R discriminates $D \circ \varphi$. The size of operation E is the size of **domain**(E).
Output: A list of numbers i for which $R(i)$ has changed.

In Section 4, we prove:

Theorem 1 Multiple BDD Online Refinement can be solved in time $O(n \min(k, \log n) + k)$, where n is the number of nodes in the BDDs and k is the total size of all operations. Thus, if n also bounds k, then the algorithm is $O(n \log n)$.

3 A Theoretical Framework for BDDs

This section develops a theory of how BDDs arise as fixed points. The main insight is the formulation of composed operators that refine partitions and that carry out path compression. We show that canonical BDDs arise as fixed points of such operators; in particular, a weak operator is exhibited that calculates the proper fixed point even as it seemingly loses information.

The Canonical BDD We define the canonical BDD for function $\psi : \mathbb{B}^{\nu} \to D$, where D is finite, as follows. A *partial assignment* $\mathbf{u}$ *from* i *to* j is a truth assignment to variables $x_i, \ldots, x_j$. The partial assignment $\mathbf{u}$ may be narrowed to a partial assignment from i' to j', where $i \leq i' \leq j' \leq j$. It is denoted $\mathbf{u}[i'..j']$. If only a prefix of $\mathbf{u}$ up to $i' - 1$ is cut off, we write $\mathbf{u}[i'..]$. An *extension* $\mathbf{v}$ *of* $\mathbf{u}$ *up to* j is a partial assignment from $i+1$ to j. A *full* extension is one that assigns up to $\nu - 1$. For any assignment prefix $\mathbf{u}$ up to i, we may consider the *residue* function $\psi_{\mathbf{u}} : \mathbf{v}' \mapsto \psi(\mathbf{u}\mathbf{v}')$, where $\mathbf{v}'$ is a full extension. Define $\mathbf{u} \sim_{\psi} \mathbf{u}'$ if $\psi_{\mathbf{u}} = \psi_{\mathbf{u}'}$. The equivalence class of $\mathbf{u}$ is denoted $[\mathbf{u}]_{\psi}$. In particular, if $\mathbf{u} \sim_{\psi} \mathbf{u}'$ then $\mathbf{u}$ and $\mathbf{u}'$ are assignment prefixes up to some i, which is called the *index* of the equivalence class $[\mathbf{u}]_{\psi} = [\mathbf{u}']_{\psi}$.

The equivalence classes of $\sim_{\psi}$ correspond to the states of a canonical automaton that upon reading a value assignment is in a state designating the value of ψ.

The *path compression* of BDDs can now be understood as a least fixed point calculation that involves coalescing equivalence classes. If $[\mathbf{u}0]_{\psi} = [\mathbf{u}1]_{\psi}$, then

$[\mathbf{u}]_\psi$ and $[\mathbf{u}0]_\psi = [\mathbf{u}1]_\psi$ are coalesced. Note that if also $[\mathbf{v}0]_\psi = [\mathbf{v}1_\psi]$ for some $\mathbf{v}$, then this identity still holds after $[\mathbf{u}]_\psi$ and $[\mathbf{u}0]_\psi = [\mathbf{u}1]_\psi$ are coalesced. Thus there is a unique least fixed point reached by repeatedly coalescing $\sim_\psi$ classes. The equivalence classes of the resulting partition $\approx_\psi$ is the *canonical* BDD for ψ. Each such new equivalence class M consists of a number of equivalence classes of $\sim_\psi$. When M contains internal nodes, the index $M.i$ of M is defined as the highest index of an old class. It can be seen that there is at most one old class in M of highest index. The high successor $M.1$, defined if the index is less than ν, is the equivalence class of $\mathbf{u} \cdot 1$, where $\mathbf{u}$ is a prefix of maximal length in M. The low successor is defined similarly.

Lemma 1 Consider i and assignment prefix $\mathbf{u}$ up to $j < i$. The following are equivalent:

1. The residue function $\psi_{\mathbf{u}\cdot\mathbf{v}}$ is the same function for all extensions $\mathbf{v}$ up to i.
2. $u \approx_\psi u \cdot v$ for all extensions v up to i.
3. $u \approx_\psi u \cdot v$ for some extension v up to i.

The equivalence $\equiv_\varphi$, which is derived from the BDD viewed as a graph, refines the equivalence $\approx_\varphi$, which is derived from the function represented by the BDD.

Lemma 2 $\equiv_\varphi$ refines $\approx_\varphi$.

Partitions of BDD Nodes A *partition* $\mathcal{P}$ of a BDD φ is a set of non-empty, disjoint subsets or *blocks* of nodes, whose union is the set of all nodes. Alternatively, $\mathcal{P}$ may be viewed as an equivalence relation $\equiv_\mathcal{P}$ defined by $v \equiv_\mathcal{P} v'$ iff $\exists B \in \mathcal{P} : v, v' \in B$. Since any assignment prefix $\mathbf{u}$ leads to a unique node $\varphi, \equiv_\mathcal{P}$ induces an equivalence relation on assignment prefixes that is also denoted $\equiv_\mathcal{P}$.

To simplify matters, we assume in the following that *all partitions are over the same BDD* φ. Also for simplicity, we shall often write $\mathcal{P}$ for $\equiv_\mathcal{P}$.

For a partition $\mathcal{Q}$, we may we define a discriminator labeling D of the leaves of φ such that for leaves v and v', $D(v) = D(v')$ iff $v \equiv_\mathcal{Q} v'$. The canonical BDD for the function $D \circ \phi$ is denoted $\approx_\mathcal{Q}$. Note that this BDD is dependent only on the partition of the leaves defined by $\mathcal{Q}$—not the partition of internal nodes. These distinctions will be further elaborated on in the next section. Note here that the partition of internal nodes may not even induce a BDD on equivalence classes. We usually regard the canonical BDD $\approx_\mathcal{Q}$ as a partition of the nodes of φ.

Decision Partitions An important part of our algorithm is to work with partitions that become refinements of canonical partitions only after certain nodes have been moved around.

A node v is a *decision node* if it is a leaf or it has at least one successor outside its own block. Any other node is *redundant*. A *decision partition* $\mathcal{M}$ of a partition $\mathcal{Q}$ specifies a partition of the decision nodes of each block B in $\mathcal{Q}$ into *decision blocks* such that any decision block M either contains internal nodes of the same index or contains only leaves. If for each block B, all decision nodes of B are gathered in just one decision block, then $\mathcal{M}$ is said to be the *stable* decision partition.

The *Split* Operator We say that the *behavior* of an internal node v with respect to a partition $\mathcal{Q}$ is the pair $([v \cdot 0]_{\mathcal{Q}}, [v \cdot 1]_{\mathcal{Q}})$; the behavior of a leaf v is just itself.

Given $\mathcal{Q}$, we can form a decision partition $\mathcal{M} = \mathit{Split}(\mathcal{Q})$ as follows. For every block B, put all decision nodes v with the same index and the same behavior in the same decision block. All leaves in B are also put into a decision block. Formally, $\mathcal{M}$ is defined as

$$\begin{aligned}\{M \neq \emptyset \mid \exists B, B_0, B_1 \in \mathcal{M} : \exists i :\\ &M = \{v \mid v \in B \text{ and } v \text{ is a leaf}\} \text{ or}\\ &M = \{v \mid v.i = i \text{ and } v \in B,\ v0 \in B_0, \text{ and } v1 \in B_1\}\}\end{aligned}$$

Partition $\mathcal{Q}$ is *stable* if *Split*($\mathcal{Q}$) is the stable decision partition.

Note that if $\mathcal{Q}$ is stable, then both successors of any internal decision node are outside its own block—for if some block B contained a decision node v with only one successor not in B, then by following the other successor, we would reach another node in B (which may be a decision node or a redundant node); by continuing, we would eventually reach a decision node in B that is either a leaf or both of whose successors are outside B and in both cases, this node would have a different behavior than that of v, and that would contradict that $\mathcal{Q}$ is stable.

Note also that $\approx_{\mathcal{Q}}$ is stable.

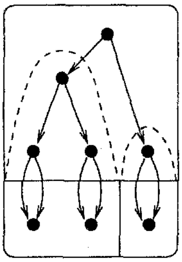
The *Grow* Operator Let $\mathcal{M}$ be a decision partition for a partition $\mathcal{Q}$. For any node v in a block B and any extension $\mathbf{u}$, there will be a first decision node w in some decision block M along the path induced by following nodes from v according to u. In this case, we say that extension $\mathbf{u}$ from v *hits* M. In particular, if $v \in M$, then any extension hits M.

If M is a decision block of B, then its *closure*, denoted $\mathit{Cl}(\mathcal{Q}, M)$, is the set of nodes in B all of whose extensions hit M. This set can also be defined inductively by *growing* the decision block: initially, let $\mathit{Cl}(\mathcal{Q}, M)$ be the decision block and add any node both of whose successors are in $\mathit{Cl}(\mathcal{Q}, M)$ until there are no more such nodes. Note that if M and M' are different decision blocks, then $\mathit{Cl}(\mathcal{Q}, M)$ and $\mathit{Cl}(\mathcal{Q}, M')$ are disjoint.

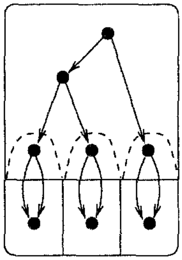
For each block B, let the *remainder*, denoted $\mathit{Rem}(\mathcal{Q}, \mathcal{M}, B)$, be defined as B minus all nodes in $\mathit{Cl}(M)$, where M is contained in B, i.e. $\mathit{Rem}(\mathcal{Q}, \mathcal{M}, B) = B \setminus \bigcup_{M \in \mathcal{M}, M \subseteq B} \mathit{Cl}(\mathcal{Q}, M)$. Then, all sets $\mathit{Cl}(\mathcal{Q}, M), M \in \mathcal{M}$, together with $\mathit{Rem}(\mathcal{Q}, \mathcal{M}, B), B \in \mathcal{Q}$, form a partition, called $\mathit{Grow}(\mathcal{Q}, \mathcal{M})$. Since, $\mathit{Grow}(\mathcal{Q}, \mathcal{M})$ is gotten from $\mathcal{Q}$ by carving out closures of decision blocks, $\mathit{Grow}(\mathcal{Q}, \mathcal{M})$ refines $\mathcal{Q}$. Note that $\mathcal{Q}$ is stable if and only if it is a fixed point under $\mathit{Grow} \circ \mathit{Split}$.

Sometimes it is convenient to assume that *Split*($\mathcal{Q}$) really stands for ($\mathcal{Q}$, *Split*($\mathcal{Q}$)). Then, we refer to the composed operator $Grow \circ Split(\mathcal{Q})$ as an abbreviation of $Grow(\mathcal{Q}, Split(\mathcal{Q}))$.

It is not necessarily the case that if $\mathcal{P}$ refines $\mathcal{Q}$, then $Grow \circ Split(\mathcal{P})$ refines $Grow \circ Split(\mathcal{Q})$. This non-monotonicity can be illustrated by the following example, where the original partitions are shown in solid lines and the additional subdivisions introduced by the $Grow \circ Split$ operator are shown in dotted lines:



$\mathcal{Q}$ and $Grow \circ Split(\mathcal{Q})$



$\mathcal{P}$ and $Grow \circ Split(\mathcal{P})$

Here, $\mathcal{P}$ refines $\mathcal{Q}$, but the two top-most nodes are equivalent in $Grow \circ Split(\mathcal{P})$, but not in $Grow \circ Split(\mathcal{Q})$.

Lemma 3 Let $\mathcal{P}$ be a stable partition and let $v \equiv_{\mathcal{P}} v'$, where v is of index i and v' of index j with $i \leq j$. Then for any extension $\mathbf{u}$ from v, $v \cdot \mathbf{u} \equiv_{\mathcal{P}} v' \cdot \mathbf{u}[j..]$.

Let $\mathcal{M}$ be a decision partition of $\mathcal{Q}$. We say that $\mathcal{P}$ *refines* $\mathcal{M}$ if whenever v and v' in are different decision blocks of $\mathcal{M}$, they are in different blocks of $\mathcal{P}$.

Lemma 4 Let stable $\mathcal{P}$ refine $\mathcal{Q}$ and $\mathcal{M}$, where $\mathcal{M}$ is a decision partition of $\mathcal{Q}$. Then $\mathcal{P}$ refines $Grow(\mathcal{Q}, \mathcal{M})$.

Lemma 5 If stable $\mathcal{P}$ refines $\mathcal{Q}$, then $\mathcal{P}$ refines $Split(\mathcal{Q})$.

Proposition 1 If stable $\mathcal{P}$ refines $\mathcal{Q}$, then $\mathcal{P}$ refines $Grow \circ Split(\mathcal{Q})$.

Proposition 2 If $\mathcal{Q} = Grow \circ Split(\mathcal{Q})$, then $\mathcal{Q}$ refines $\approx_{\mathcal{Q}}$.

Proposition 3 If $\approx_{\mathcal{Q}}$ refines $\mathcal{Q}$ and if $\mathcal{Q}' = (Grow \circ Split)^i(\mathcal{Q})$ is stable, then $\mathcal{Q}'$ is $\approx_{\mathcal{Q}}$.

The *CGrow* Operator The *CGrow* operator is defined as $Grow(\mathcal{Q}, \mathcal{M})$ except that for each block B of $\mathcal{Q}$, $Rem(\mathcal{Q}, \mathcal{M}, B)$ may or may not be coalesced with some designated $Cl(\mathcal{Q}, M)$, where M is a decision block in B. Thus the operation is not fully specified, but whether coalescing takes place or not and with which $Cl(\mathcal{Q}, M)$ will be inconsequential for establishing the following general properties. Note that $Grow \circ Split(\mathcal{Q})$ refines $CGrow \circ Split(\mathcal{Q})$. Even though information is dropped by *CGrow*, a fixed point involving *CGrow* is also a fixed point involving *Grow*:

Proposition 4 If $CGrow \circ Split(Q) = Q$, then $Grow \circ Split(Q) = Q$.

Theorem 2 If $\approx_Q$ refines Q and if $Q' = (CGrow \circ Split)^i(Q)$ is stable, then Q' is $\approx_Q$.

Our concept of leaf partition can then be understood as a decision partition $\mathcal{E}$ of the current canonical partition Q. The only non-trivial decision blocks of a leaf partition are those that contain leaves. A canonical equivalence relation $\approx_{\mathcal{E}}$ is defined as before for $\approx_Q$.

Theorem 2 then can be formulated:

Theorem 2′ If $\approx_{\mathcal{E}}$ refines Q and if $Q' = CGrow \circ (Split \circ CGrow)^i(Q, \mathcal{E})$ is stable, then Q' is the canonical partition $\approx_{\mathcal{E}}$.

4 Online Algorithm

The online problem in Section 2 can be solved by maintaining the canonical partition by means of a node discriminator for all nodes, not only the roots. In this way, we may focus on the refinement problem for a single BDD, since multiple BDDs can be embedded within a single one by introducing dummy variables near the root. The modified problem is:

Single BDD Online Refinement Problem
Input: A BDD φ with leaves $\mathcal{L}$ and constant discriminator D.
Maintained : For each node v, the discriminator value $D(v)$ is maintained so that D expresses the canonical partition $\approx_{\mathcal{E}}$.
Update: A partial mapping $E : \mathcal{L} \to \mathbb{N}$ such that **range**$(E) \cap$ **range**$(D) = \{\}$. E and the current partition of leaves determine a leaf partition $\mathcal{E}$.
Output: A list of nodes for which $D(v)$ has changed.

As an example, consider the BDD in Figure 1. The leaf partition at this stage has been refined into two decision blocks. The canonical partition with respect to this decision partition is indicated by dotted lines. An update operation E might split the two leaves in the left most block, and as a result, the four nodes in the left, bottom corner would each become a singleton equivalence class.

The basic problem encountered when trying to construct a fast algorithm is that after nodes have been split, it is necessary to calculate equivalence under path compression—corresponding to our notion of growing decision blocks. There is now evident way of carrying out the grow phase, which must proceed bottom-up, without touching nodes more than $\Theta(\log n)$ times. Our notion of coalesced growth opens an escape hatch that allows the process to be halted at certain critical moments.

Our algorithm works as follows. The canonical partition $\approx_{\mathcal{E}}$ induced by the leaf partition $\mathcal{E}$ refines the current partition Q expressed by D. Therefore, according to Theorem 2′, we can apply the combined operator $Split \circ CGrow$ until a new fixed point Q' is reached. Then, Q' is the canonical partition $\approx_{\mathcal{E}}$.

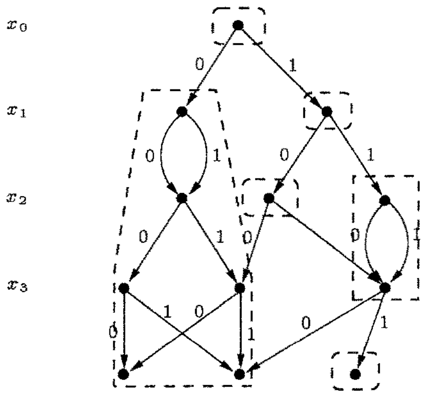


Figure 1. A canonical BDD

To make this abstract description into an algorithm, we must choose data structures and explain how the split and grow operations are implemented. We also must explain how we choose the coalescing of blocks in *CGrow*.

Each discriminator value d represents a block that we denote by d. We maintain a doubly-linked list $L(d)$ of all v in d. A decision partition is specified for a block d_{old} by *explicit* decision blocks and a *implicit* decision block. They are carved out of the block d_{old} as follows. Each explicit decision block is represented by a discriminator value d, and all nodes in the decision block d are placed in the list $L(d)$, which is carved out of $L(d_{old})$. Later, when the decision blocks are grown, these discriminator values will denote their closures. In addition, the implicit decision block consists of all decision nodes in the block not appearing in an explicit decision block. The algorithm will in a gradual fashion convert the implicit decision nodes to explicit ones carrying some distinct discriminator value $d_{implicit}$ reserved for the explicit version of the implicit block.

The algorithm uses a mapping $new(d_{old})$ that records the set of discriminator values for the decision blocks in d_{old}.

Initially, we call the *CGrow* algorithm with decision blocks of leaves and *new* initialized according to E.

The *CGrow* phase is implemented for each decision block $L(d)$ by adding the nodes in $L(d_{old})$ for which both successors are already in $L(d)$; such nodes are removed from $L(d_{old})$. To locate nodes that should be considered for inclusion in a closure, we assume that the BDD is equipped with a backwards pointer structure such that the parents of any node can be sequentially accessed. This process of exploring parents is done in a tightly controlled manner according to the sizes of the lists $L(d)$ for $d \in new(d_{old})$. When a parent has been explored from both the left and right successor, and both are in the same closure, then the parent is moved into this closure as well. The exploration of a closure finishes when all parents of all nodes in the closure have been explored.

The *CGrow* phase returns a list of all nodes possessing a successor whose discriminator has changed. These nodes are the explicit decision nodes of the next iteration.

The *Split* algorithm calculates the new discriminator of the nodes in this list according to their behavior. It also calculates the value of *new*.

Main Idea

The main idea behind the *CGrow* phase is that all unfinished closures are grown in parallel steps, where each step consists of exploring yet another parent of a node in the closure until either (a) a closure becomes too big, say half the size of d_{old}, or (b) until only one closure is unfinished or (c) until all closures are finished. In case (a) and (b), the closure in question is coalesced with the remainder by moving nodes back to d_{old}. (If the conversion of implicit decision nodes is not yet finished, the step for the implicit block is simply to convert another node to $L_{d_{implicit}}$. When all nodes have been converted, this decision block is treated as an ordinary one.) In case (a), all remaining closures are then finished and they will all be small since a big one already was found. In case (b), all closures, possibly except the last one (if it was finished), will by the absence of the condition in case (a) be small.

In case (a), the work involved in building the aborted closure can be charged to a small, finished closure. For this argument to be correct, it is crucial that the work done is the same (to within a constant factor) for all the closures grown in parallel.

In case (b), there may be no small, finished closure to charge the wasted work to. This situation occurs when there is only one decision block to begin with. In this case, the work involved will be proportional to the size of the decision block, and it can be assumed to be part of the work involved in building the decision block. The algorithm makes sure that the original discriminating value d_{old} of the whole block is maintained despite a possible new value assigned to the decision block. In this way, only blocks that are really split may result in further splitting.

In case (c), all blocks will be small. The work done in building a closure is not proportional to the size of the closure, since each parallel step consists of exploring a parent (of which there may be unboundedly many). But each parent has only two successors, and so, the work of visiting the parent can be charged to the closure of the child from which it is explored (unless the work is attributed to another block as a result of the abandonment of a closure calculation). Thus, every time a parent is explored from the same successor, it will be done when the resulting closure the successor resides in is at most half as big as the last time.

In the full paper, we provide a more detailed description, a complexity analysis, and a discussion about how hashing can be avoided.

Theorem 1′ The Single BDD Online Problem can be solved in time $O(n \min(k, \log n) + k)$, where n is the number of nodes in the BDDs and k is the total size of all operations. Thus, if n also bounds k, then the algorithm is $O(n \log n)$.

Theorem 1 follows from Theorem 1′

5 Minimizing BDD Represented Automata

In the full paper, we explain how to obtain:

Corollary 1 Minimization is $O(n \log n)$ for BDD-represented automata, where n bounds the number of states and the number of BDD nodes.

Acknowledgments

Thanks to Robert Paige and Theis Rauhe for their careful reading of an earlier version of this paper, for pointing out errors, and for exploring the possible existence of a simpler $n \log n$ BDD online refinement algorithm. The example of non-monotonicity of the composed operator in Section 3 was suggested by Robert Paige to illustrate an error in the earlier version. Michael Yannakakis kindly pointed out the reference [2].

References

1. D. Basin and N. Klarlund. Beyond the finite in hardware verification. Submitted. Extended version of: "Hardware verification using monadic second-order logic," *CAV '95*, LNCS 939, 1996.
2. Norbert Blum. An $o(n \log n)$ implementation of the standard mothod for minimizing n-state finite automata. *Information Processing Letters*, 1996.
3. R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing surveys*, 24(3):293–318, September 1992.
4. A. Cardon and M. Crochemore. Partitioning a graph in $O(|A| \log_2 |V|)$. *TCS*, 19:85–98, 1982.
5. Aarti Gupta. *Inductive Boolean function manipulation.* PhD thesis, Carnegie Mellon University, 1994. CMU-CS-94-208.
6. Aarti Gupta and Allan L. Fisher. Representation and symbolic manipulation of linearly inductive boolean functions. In *Proceedings of the IEEE International Conference on Computer-Aided Design*, pages 192–199. IEEE Computer Society Press, 1993.
7. J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1996. Also available through http://www.brics.dk/~klarlund/MonaFido/papers.html.
8. J. Hopcroft. An $n \log n$ algorithm for minimizing states in a finite automaton. In Z. Kohavi and Paz A., editors, *Theory of machines and computations*, pages 189–196. Academic Press, 1971.
9. D. Lee and M. Yannakakis. Online minimization of transition systems. In *Proc. STOC*, pages 264–274. ACM, 1992.
10. H-T. Liaw and C-S. Lin. On the OBDD-representation of general Boolean functions. *IEEE Trans. on Computers*, C-41(6):661–664, 1992.
11. R. Paige and R. Tarjan. Three efficient algorithms based on partition refinement. *SIAM Journal of Computing*, 16(6), 1987.
12. D. Sieling and I. Wegener. Reduction of OBDDs in linear time. *IPL*, 48:139–144, 1993.