Binary Decision Graphs

Laurent Mauborgne

LIENS – DMI, École Normale Supérieure, 45 rue d'Ulm, 75 230 Paris cedex 05, France
Tel: (+33) 01 44 32 20 66; Email: Laurent.Mauborgne@ens.fr
WWW home page: http://www.dmi.ens.fr/~mauborgn/

Abstract. Binary Decision Graphs are an extension of Binary Decision Diagrams that can represent some infinite boolean functions. Three refinements of BDGs corresponding to classes of infinite functions of increasing complexity are presented. The first one is closed by intersection and union, the second one by intersection, and the last one by all boolean operations. The first two classes give rise to a canonical representation, which, when restricted to finite functions, are the classical BDDs. The paper also gives new insights in to the notion of variable names and the possibility of sharing variable names that can be of interest in the case of finite functions.

1 Introduction

Binary Decision Diagrams (BDDs) were first introduced by Randal E. Bryant in [4]. They turned out to be very useful in many areas where manipulation of boolean functions was needed. They allowed a real breakthrough of model checking [7], they have been used successfully in artificial intelligence [14] and in program analysis [8, 13, 2, 16].

One limitation of BDDs and its variants is that they can only represent finite functions. Indeed, it induces a well-known and quite annoying restriction on model checking, and it restrains its use in program analysis. This paper explores the possibility of extending BDDs so that they can also represent infinite functions. This extension will allow the model checking of some infinite state systems or unbounded parametric systems, or the static analysis of the behavior of infinite systems, where the expression of infinite properties such as fairness is necessary.

After a presentation of our notations, sections 3 and 4 present the main ideas that allow this extension: the first idea is the possibility of sharing variable names for different entries, while preserving a sound elimination of redundant nodes. The second idea is the possibility of looping in the representation (thus the term of Binary Decision Graph (BDG) instead of Binary Decision Diagram), while preserving the uniqueness of the representation. Sections 5 to 8 present three classes of infinite boolean functions of increasing complexity corresponding to further refinements of the representation by BDGs. Only the last one is closed by all boolean operations, but the first two are representable by efficient unique BDGs, which, when used to represent finite functions, give classical BDDs. Both

classes have results on approximation properties that can easily be exploited in abstract interpretation [9], a fact that is very promising on their usefulness in program analysis.

2 Boolean Functions, Vectors and Words

Let $\mathcal{B} \stackrel{\text{def}}{=} \{\texttt{true}, \texttt{false}\}$ be the set of boolean values. $\mathcal{B}^n$ denotes the set of boolean vectors of size n. A finite boolean function is a function of $\mathcal{B}^n \to \mathcal{B}$. $\mathcal{B}^\omega$ denotes the set of infinite boolean vectors. An infinite boolean function is a function of $\mathcal{B}^\omega \to \mathcal{B}$.

A boolean function f is entirely characterized by a set of vectors, which is defined as $\{\, u \mid f(u) = \texttt{true}\}$. Thus, we will sometime write $u \in f$ for $f(u) = \texttt{true}$. This characterization is interesting to distinguish between the variables of the functions and their position in the function, which we call its entries. If $f : \mathcal{B}^n \to \mathcal{B}$, then the entries of f are the integers between 0 and $n-1$. If $f : \mathcal{B}^\omega \to \mathcal{B}$, then the entries of f are $\mathbb{N}$, the set of all natural numbers. We write $u_{(i)}$ for the projection of u on its i^{th} value. Given a set I of entries, $u_{(I)}$ denotes the subvector of u with I as its set of entries. The *restriction* of f according to one of its entries i and to the boolean value b is denoted $f|_{i \leftarrow b}$ and is defined as the set of vectors: $\left\{ u \mid \exists v \in f,\ v_{(i)} = b \text{ and } v_{(\{\, j \mid j \neq i\})} = u \right\}$. A special case is when $i = 0$. In this case, we simply write $f(b)$. We extend this notation to any vector whose size is smaller than the vectors in f.

It is sometime convenient to consider a boolean vector as a word over $\mathcal{B}^*$. It allows the use of concatenation of vectors. If u is a finite vector and v a vector, the vector $u.v$ corresponds to the concatenation of the words equivalent to the vectors u and v. The size of a vector u is written $|u|$. The empty word is denoted ε. We define formally the notation $f(u)$:

$$f(u) \stackrel{\text{def}}{=} \{\, v \mid u.v \in f\} \text{ if } f : \mathcal{B}^\omega \to \mathcal{B} \text{ or } f : \mathcal{B}^n \to \mathcal{B} \text{ and } |u| < n$$

So, if for all vectors f is false, we can write $f = \emptyset$.

We extend the concatenation to sets of vectors: for example, if f is a set of vectors and u a vector, $u.f \stackrel{\text{def}}{=} \{\, u.v \mid v \in f\}$.

To improve the clarity of the article, we adopt the following conventions:

a, b, c	represent boolean values,
e, f, g	represent boolean functions,
u, v, w	represent vectors. In a context where we have infinite vectors, represent finite vectors,
α, β, γ	represent infinite vectors,
x, y, z	represent variables (or entry names),
r, s, t	represent binary trees,
i, j, k, n	represent natural numbers.

In the description of vectors, to reduce the size of the description, we will write 0 for `false`, and 1 for `true`.

3 Entry Names

In classical BDDs, boolean functions are described by boolean expressions using variable names corresponding to the different entries of the BDDs. The Binary Decision Diagrams are ordered, so that the ordering imposed on the variables follows the order of the entries of the function. The variable name of rank i is called the name of the entry i. In many cases, the variable names correspond to some entities related by the boolean function. A consequence is that we can have the same information while changing the ordering on the variable names and the boolean functions that bind them (so that a given entry always corresponds to the same variable name, and the ordering on the variable names is the same as the entries ordering). Thus the different optimizations depend on the choice of this ordering [10, 12].

3.1 Equivalent Entries

In the case of infinite functions, we cannot assign a different variable name to all the entries of the boolean function, because we want a finite representation. The idea is that the set of variable names associated with a given function is finite and to achieve that, some entries can share the same name. However, not every entry can share the same name: to share the same name, two entries must be in a sense equivalent.

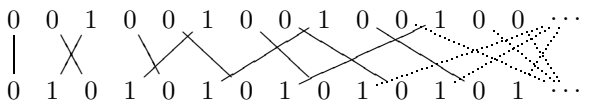
A permutation is a bijection of $\mathbb{N} \to \mathbb{N}$. If $I \subset \mathbb{N}$, a permutation of the entries in I is a permutation σ such that $\forall i \notin I$, $\sigma(i) = i$. A permutation σ defines a function from vectors to vectors, $\overrightarrow{\sigma}$, defined by : $\overrightarrow{\sigma}(u)_{(i)} = u_{(\sigma(i))}$ for all i entry of u.

Definition 1 (Equivalent entries). *Let f be a boolean function. The entries contained in the set $I \subset \mathbb{N}$ are equivalent if and only if whatever the permutation σ of the entries in I, whatever the vector $u \in dom(f)$, $f(u) = f(\overrightarrow{\sigma}(u))$*

There are two ideas underlying the definition of equivalent entries: the restriction according to any equivalent entry is the same, so $f|_{x \leftarrow b}$, where x is an entry name, is not ambiguous; and whatever the ordering of the equivalent entries, the function is the same. The following example shows that this property imposes that we allow infinite permutations.

Example 1. Consider the infinite function that is true on any infinite vector containing two consecutive 0s infinitely many times. This function contains $(001)^\omega$ but not $(01)^\omega$. If we only considered finite permutations, that is permutations generated from finite exchange of entries, then all entries of this function are equivalent. But there is an infinite permutation that transforms $(001)^\omega$ into $(01)^\omega$:

$\sigma(3 \times k) = 2 \times k$
$\sigma(3 \times k + 1) = 4 \times k + 1$
$\sigma(3 \times k + 2) = 4 \times k + 3$



The meaning of this substitution is that, if a function with all entries equivalent contains $(001)^\omega$ then there is a way of giving the values of $(01)^\omega$ such that it is accepted by the function. Concerning our function, it is consistent to forbid the equivalence of all entries.

As an immediate consequence, we have the following properties for functions f where all entries are equivalent:

Proposition 1. *Let v be a word, and b a boolean value in v. Then $v^\omega \in f$ if and only if $(vb)^\omega \in f$.*

Proposition 2. *Let α be a word where a boolean value b appears infinitely often. Then $\alpha \in f$ if and only if $b.\alpha \in f$.*

Proof. In both cases, we have an infinite permutation of the entries that transforms the first vector into the other one. In the second case, we just have to shift the b's of $b.\alpha$ to the right, each b going to the entry of the next one. In the first case, we keep shifting by one more b for each $v.b$. □

3.2 Equivalent Vectors of Entries

In order to accept more functions, we extend the notion of equivalent entries to equivalent vectors of entries. We just consider as one entry a whole set of entries of the form $\{ i \in \mathbb{N} \mid k \leq i < k + n\}$. It will allow the iteration over whole vectors of entries. The set of equivalent entries is described by a set I of indexes and a length n such that $\forall k \in I$, $\forall i$ such that $k < i < k + n$, $i \notin I$. A substitution σ over such a set is such that $\forall k \in I$, $\sigma(k) \in I$, and $\forall i < n$, $\sigma(k + i) = \sigma(k) + i$. For all other numbers j, $\sigma(j) = j$.

Definition 2 (Equivalent Vectors of Entries). *Let f be a boolean function. The vectors of entries contained in the set I with length n are equivalent if and only if whatever the permutation σ of the entries in I, whatever the vector $u \in dom(f)$, $f(u) = f(\overrightarrow{\sigma}(u))$*

Two entries can have the same name if and only if they are at the same position in a set of equivalent vectors of entries. We suppose in the sequel that every boolean function comes with such a naming of the entries. We will write $\text{name}_f(i)$ for the name of the entry i of function f. To simplify the presentation and the proofs, we will only consider simple equivalent entries in the sequel of the article, but the results extend easily to equivalent vectors of entries.

3.3 Equivalent Entries and Redundant Choices

Redundant choices are used in BDDs to reduce the size of the representation. The good news is that giving the same name to equivalent entries is compatible with the elimination of redundant choices. There is a redundant choice at a subvector u of f if and only if $f(u.0) = f(u.1)$.

Theorem 1. *Let f be a boolean function, and u be a vector such that $f(u.0) = f(u.1)$. Then, whatever v such that $name_f(|u.v|) = name_f(|u|)$, $f(u.v.0) = f(u.v.1)$.*

Proof. $v = a.w$. We have $f(u.a.w.0) = f(u.0.w.a)$ because of the equivalence of the entries. $f(u.0.w.a) = f(u.1.w.a)$ by redundancy of the choice, and $f(u.1.w.a) = f(u.a.w.1)$ by equivalence of the entries. Thus $f(u.v.0) = f(u.v.1)$. □

3.4 Periodicity of the Entries

In order to be finitely representable, we impose some regularity to the entry names. The entry names are said to be *periodic* if and only if there is a period k on the entry names, that is, for all i, the name of $i + k$ is the same as the name i. The entry names are said to be *ultimately periodic* if, after some point, they are periodic. Throughout the paper, we will assume that the entry names are ultimately periodic.

4 Decision Trees

4.1 Finite Decision Trees

BDDs are based on decision trees. A decision tree is a structured representation based on Shannon's expansion theorem: $f = 0.f(0) \cup 1.f(1)$. This observation is the basis of a decision procedure: to know whether a given vector is in a function, we look at its first value. If it is a 0, we iterate the process on the rest of the vector and $f(0)$, and if it is a 1, we iterate on the rest of the vector and $f(1)$. This procedure can be represented by a binary tree labeled by the entry names, and with either `true` or `false` at the leaves.

We define a labeled binary tree t as a partial function of $\{0,1\}^* \to L$ where L is the set of labels, and such that whatever $u.v \in dom(t)$, $u \in dom(t)$. The subtree of t rooted at u is the tree denoted $t_{[u]}$ of domain $\{\, v \mid u.v \in dom(t)\}$, and defined as $t_{[u]}(v) \stackrel{\text{def}}{=} t(u.v)$. The decision tree defined by a boolean function $f : \mathcal{B}^n \to \mathcal{B}$ is the binary tree of domain $\bigcup_{k \leq n} \{0,1\}^k$, such that if $|v| = n$, then $t(v) = f(v)$, and if $|v| < n$, $t(v) = \text{name}_f(|v|)$.

Example 2. Let $f = \{000, 011, 111\}$. If we associate the variable names x to entry 0, y to entry 1 and z to entry 2, then f can be described by the formula: $(y \wedge z) \vee (\neg x \wedge \neg y \wedge \neg z)$. The decision tree for f is displayed in Fig. 1.

4.2 Semantics of the Decision Trees

The decision tree of a boolean function is used as a guide for a decision process that decides the value of the function on a given vector. If t is the decision tree associated with the function f, then to decide whether the vector u is in f, we "read" the first value of u. Say $u = b.v$. If b is a 0, we iterate on v, $t_{[0]}$, and

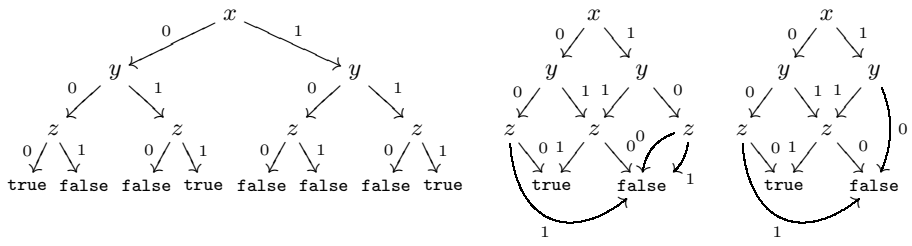
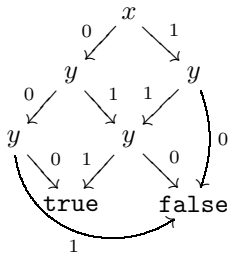


Fig. 1. The decision tree, the decision tree with shared subtrees, and the BDD.

if it is a 1, we iterate on v, $t_{[1]}$. The entire decision process goes through the tree, following the path defined by u, and the result of the decision process is the value of the leaf, $t(u)$. Binary Decision Diagrams are based on two remarks: first we can represent any tree in a form where equivalent subtrees are shared (a directed acyclic graph), and second if a choice is redundant, then we can jump it. The second remark modifies slightly the decision process: we must have separate information on the entries of a function. For example, we can have the sequence of the entry names, or equivalently if all entry names are different, an ordering on the entry names. In this way, we can keep track of the current entry that is read from u. If it is "before" the entry named $t(\varepsilon)$, we can skip the first value of u and iterate on v, t.

Example 2 (continued). The two steps of sharing equivalent subtrees and eliminating redundant nodes are shown in Fig. 1. The decision process on the vector 101 can reach `false` after reading the first 1 and the first 0. The entry names of the BDD are represented by $x < y < z$, or equivalently by xyz. If we realize that the entry 1 and the entry 2 are equivalent, we can give the same name y to both entries. Then the BDD becomes:



with entry names described as xyy. It is easy to see that this description will lead to a more efficient algorithm to compute $f|_{2\leftarrow 0}$ which correspond to $f|_{z\leftarrow 0}$ with the entry names xyz, and to $f|_{y\leftarrow 0}$ with the entry names xyy.

It is an established fact [4], that given a boolean function and a naming of the entries with all names different, such a representation is unique, leading to tautological equivalence testing. From Theorem 1, we can add that this representation is still unique if the naming of the entries respect the equivalences of the entries.

4.3 Infinite Trees

To extend this definition to infinite boolean functions, there are two problems: for all $\alpha \in f$, $\alpha \notin dom(t)$, because binary trees domains are limited to finite words. This is the problem of the *infinite behavior* of the function. We can represent $t(v)$ for all v prefix of a vector in f, but then the tree is infinite. This is the second problem, treated in this section: how to represent an infinite tree.

As we have seen, a BDD is a decision tree on which we have performed two operations: first the sharing of equivalent subtrees, second the elimination of redundant choices. In fact, following this process would be too inefficient, and when manipulating BDDs, these operation are performed incrementally: each time we build a tree $\overset{x}{\underset{t\ t}{\swarrow\!\searrow}}$, we return t, and each time we build another tree $\overset{x}{\underset{t_0\ t_1}{\swarrow\!\searrow}}$, we first look if the tree has already been encountered, through a hash table for example, and if it is the case, we return the tree already encountered, if not we add it in the table.

The same operations, albeit a little more complex, can be performed to represent an infinite tree with maximal sharing of its subtrees. First we only represent *regular* trees, that is trees with a finite number of distinct subtrees. The only difference with finite trees, which are represented by directed acyclic graphs, is that infinite trees are represented by directed graphs that do contain cycles. The added complexity introduced by the cycles is not intractable, and efficient incremental algorithms can be devised. The ideas are the following: when we are not in a cycle, the algorithm is the same as in the finite case. When we isolate a strongly connected subgraph (a "cycle"), we first see if this cycle is not the unfolding of another cycle that is reachable from the subgraph. If it is the case, we return this other cycle (we fold the subgraph on the cycle). If not, we reduce the subgraph to an equivalent one with maximal sharing, and then we compute unique keys for the subgraph, so that we can see if it had already been encountered, or so that we can recognize it in the future. We have one key for each node of the subgraph. The detailed algorithms and their proofs can be found in [17].

Examples of infinite trees represented this way will be displayed in the next sections. The trees will be progressively enriched so that the BDGs represent wider classes of infinite functions.

5 Basic BDGs: Open Infinite Functions

If we just extend BDDs with the possibility of sharing entry names and the possibility of using infinite trees, we can already represent open infinite functions. Our first restriction though, is that the decision tree is representable, that is, it is regular.

Definition 3. *Let f be a boolean function. f is said to be* prefix regular *if and only if the number of distinct $f(u)$ is finite.*

Because we will have only one possible representation for a given function, and $f(u)$ corresponds to $t(u)$ if t represents f, it means that t is regular.

5.1 Open Functions and Closed Functions

Definition 4 (Open Function). *Let $f : \mathcal{B}^\omega \to \mathcal{B}$. f is said to be* open *if and only if f is prefix regular and:*

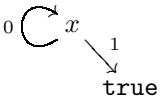
$$\forall \alpha \in f,\ \exists u \text{ such that } \alpha = u.\beta \text{ and } f(u) = \mathcal{B}^\omega$$

Recall that $f(u) = \mathcal{B}^\omega$ means that whatever γ, $f(u.\gamma) = \texttt{true}$. This definition corresponds to a choice on the meaning of an infinite decision tree labeled by entry names and with `true` and `false` as leaves. If during the decision process of a vector, we reach a `false`, then the result of f on the vector must be `false`. If we read a `true`, then the result of f on the vector must be `true`. If u is the beginning of the vector that lead to `true`, then every infinite vector beginning with u will be in f, so $f(u) = \mathcal{B}^\omega$. The last case is when we never reach `true` nor `false`. The choice is that such vectors are not in the function. By choosing that such vectors are in the function, we would have represented dually the closed functions. So, the dual of the properties on open functions apply to closed functions.

Because we chose that cycling in the decision process is rejecting, there is one new source of non-uniqueness that is not taken care of by simply sharing every subtree of the decision tree. We must also replace by `false` every cycle from which no `true` is reachable. This is easily performed while treating cycles in the representation of regular trees. With this treatment, we still have a unique representation for open functions.

Example 2 (continued). The function f can be extended to the function $g : \mathcal{B}^\omega \to \mathcal{B}$, defined as: $g(u.\alpha) = \texttt{true}$ if $u \in f$. Then g is represented with the same diagram as f, with an additional entry name z', and the entry names are $xyy(z')^\omega$.

Example 3. Let f be `true` on α if and only if α contains at least one 1. Every entry of f is equivalent, so its entry names can be described as x^ω. Since f is open, it can be represented by the following graph:



5.2 Boolean Operators

Theorem 2. *Let f and g be two open functions. Then the functions $f \wedge g$ and $f \vee g$ are open. Moreover, if $(f_i)_{i \in \mathbb{N}}$ is a family of open functions, then $\bigvee_{i \in \mathbb{N}} f_i$ is an open function.*

$f \wedge g(\alpha) \stackrel{\text{def}}{=} f(\alpha) \wedge g(\alpha)$, and $f \vee g(\alpha) \stackrel{\text{def}}{=} f(\alpha) \vee g(\alpha)$. So, if we consider f and g as sets of vectors, $f \wedge g$ is the intersection of f and g, and $f \vee g$ is the union of f and g.

Proof. Let $\alpha \in f \vee g$. There is a u such that $\alpha = u.\beta$, and either $f(u) = \mathcal{B}^\omega$ or $g(u) = \mathcal{B}^\omega$. In any case, $f \vee g(u) = \mathcal{B}^\omega$. If $\alpha \in f \wedge g$, there is u and v such that $\alpha = u.\beta$, $\alpha = v.\gamma$, and $f(u) = \mathcal{B}^\omega$ and $g(v) = \mathcal{B}^\omega$. If $|u| \leq |v|$, then $v = u.w$. So $f(v) = \mathcal{B}^\omega$. So, $f \wedge g(v) = \mathcal{B}^\omega$. If $\alpha \in \bigvee_{i \in \mathbb{N}} f_i$, then there is a least u prefix of α such that there is a i, $f_i(u) = \mathcal{B}^\omega$. We have $\bigvee_{i \in \mathbb{N}} f_i(u) = \mathcal{B}^\omega$. □

Dually, the finite union of closed functions is a closed function, and the infinite intersection of closed functions is a closed function.

Corollary 1. *Whatever the boolean function f, there is a greatest open function contained in f, and there is a least closed function containing f.*

Algorithmically, it is easy to compute the `and` or the `or` of two open functions. The algorithms are the same as in the finite case [5], with the possibility of memoizing [18], except that we must take care of cycles. When a cycle is encountered, that is when we recognize that we already have been through a pair of subtrees (s, t), we build a loop in the resulting tree.

Open functions are not closed by negation: the negation of the function that is true on all vectors containing at least one 1 is the function containing only 0^ω. Such a function is not open, because the only infinite behavior that is possible for an open function is trivial. In order to be more expressive, we introduce more infinite behaviors.

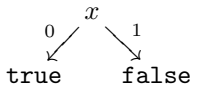
6 More Infinite Behaviors

To allow more infinite behaviors, we need to have more than one kind of loop, so that in some loop it is forbidden to stay forever, and in some others, we can. We introduce a new kind of loop: loops over open functions. This new kind of loop defines a new set of infinite behaviors, defining what we call iterative functions. Iterative functions are functions that start over again and again infinitely often. Thus entry names will have to be periodic.

Definition 5 (Iterative Function). *Let $f : \mathcal{B}^\omega \to \mathcal{B}$. f is said to be* iterative *if and only if the entry names of f are periodic, and there is an open function g such that for all α in f, there is an infinite sequence of vectors $(u_i)_{i \in \mathbb{N}}$, such that $\alpha = u_0.u_1 \ldots u_i \ldots$ and each u_i has the minimum length such that $g(u_i) = \mathcal{B}^\omega$ and $name_f(|u_i|) = name_f(0)$. We write $f = \Omega(g)$.*

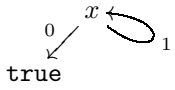
Hence an iterative function is represented by an open function. We will use the decision tree of the open function to represent the iterative function. But in the context of iterative functions, the decision tree will have a different meaning, corresponding to a slightly different decision process. The decision process is the following: we follow the decision tree in the path corresponding to the vector, but when we reach a `true`, we start again at the root of the tree. To be a success, the decision process must start again an infinite number of times.

Example 4 (Safety).



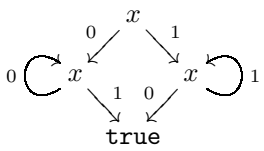
represents the function that is true on 0^ω only.

Example 5 (Liveness).



represents the function that is false only on those vectors that end with 1^ω. That is, the function is true on any vector containing an infinite number of 0's.

Example 6 (Fairness).



represents the function that is true on any vector containing an infinite number of 0's and 1's.

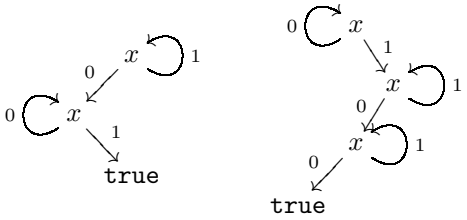
These examples show that iterative functions can be used to represent a wide variety of infinite behaviors. Note that $\emptyset$ and $\mathcal{B}^\omega$ are at the same time open and iterative. Another remark: the equivalence of the entries restraining the use of shared entry names is only applied to the iterative function, not the open function that represents the iterative function.

Theorem 3. *An iterative function is prefix regular.*

Proof. If $f = \Omega(g)$, g is open, so prefix regular. If $g(u) = g(v)$, then $f(u) = f(v)$. So the set of distinct $f(u)$ is smaller than the set of distinct $g(u)$. Thus, f is prefix regular. □

Many possible open functions can represent the same iterative function:

Example 6 (continued). The function that is true on every vector with an infinite number of 0's and 1's could also be represented by the following open function:



In fact, there is a "best" open function representing a given iterative function. If we always choose this best open function, as the representation of open function is unique, the representation of iterative function is unique too.

Theorem 4. *Let f be an iterative function. The function g which is true on the set $\{u.\alpha |\, u^\omega \in f$ and $f(u) = f\}$ is the greatest (for set inclusion) open function such that $f = \Omega(g)$*

In order to simplify the proofs, we will suppose that all entries of f are equivalent, so that there is only one entry name, and we can get rid of the test $\text{name}_f(|u|) = \text{name}_f(0)$. To reduce the problem to this case, we can use a function over larger finite sets described by $\mathcal{B}^k$, where k is a period of the entry names of f.

Lemma 1. *Let u be a vector such that $g(u) = \mathcal{B}^\omega$ and for all v prefix of u, $g(v) \neq \mathcal{B}^\omega$. Then $u^\omega \in f$ and $f(u) = f$.*

Proof (Lemma 1). Whatever α, $u.\alpha \in g$. Because of the minimality of u with respect to the property $g(u) = \mathcal{B}^\omega$, for each α, there is a v prefix of α such that $(u.v)^\omega \in f$ and $f(u.v) = f$. Let b be a boolean value in u. We choose $\alpha = b^\omega$. There is an l such that $(v.a^l)^\omega \in f$. Because all entries of f are equivalent, by Proposition 1, $v^\omega \in f$. Let $\alpha \in f$. α is infinite, so there is a boolean value a that is repeated infinitely often in α. But there is an l such that $f(v.a^l) = f$. So $v.a^l.\alpha \in f$, and by the equivalence of all entries (Proposition 2), $v.\alpha \in f$, which means that $\alpha \in f(v)$. Conversely, if $\alpha \in f(v)$, $v.a^l.\alpha \in f$, and so $\alpha \in R$. Thus $f = f(v)$.

Proof (Theorem 4). g is an open function, because whatever the element α of g, there is a u prefix of α such that $\forall \beta$, $u.\beta \in g$. f is iterative, so there is an open function g' such that $f = \Omega(g')$. $\forall \alpha \in g'$, there is a u prefix of α such that $g'(u) = \mathcal{B}^\omega$. Let u_0 be the least such u. Because $f = \Omega(g')$, ${u_0}^\omega$ is in f, and $f(\alpha) = f$. So $\alpha \in g$, which means that $g' \subset g$. To prove the theorem, we just have to prove that $f = \Omega(g)$. We will start by $f \subset \Omega(g)$, then prove $\Omega(g) \subset f$.

Let $\alpha \in f$. We suppose $\alpha \notin \Omega(g)$. If there is a u prefix of α such that $g(u) = \mathcal{B}^\omega$, let u_0 be the least such u. $\alpha = u_0.\beta$. Then $\beta \notin \Omega(g)$, but $f(u_0) = f$, so $\beta \in f$. So we can iterate on β. This iteration is finite because $\alpha \notin \Omega(g)$, so we come to a point where there is no u prefix of α such that $g(u) = \mathcal{B}^\omega$. But $\alpha \in f$, so there is a u_0 prefix of α such that $g'(u_0) = \mathcal{B}^\omega$, and $u_0^\omega \in f$ and $f(u_0) = f$, and so $g(u_0) = \mathcal{B}^\omega$, which contradicts the hypothesis. Thus $f \subset \Omega(g)$.

Let $\alpha \in \Omega(g)$. Let $(u_i)_{i \in \mathbb{N}}$ be the sequence of words such that $g(u_i) = \mathcal{B}^\omega$, $\alpha = u_0.u_1 \dots u_i \dots$ and u_i minimum. Some u_i appear infinitely often in α, and some other u_i appear only finitely often in α. Hence there is a permutation of the entries such that the result of the permutation on α is $v.\beta$, where v is the concatenation of all u_i that appear finitely in α (times the number of times they appear), and β is composed of those u_i that appear infinitely in α. By definition of $\Omega(g)$, $\beta \in \Omega(g)$, and because all entries of f are equivalent, $v.\beta$ is in f if and only if α is in f. But whatever u_i, $f(u_i) = f$ (see the lemma). So $f(v) = f$. And so α is in f if and only if β is in f. Either β contains a finite number of distinct u_i, or an infinite one.

If β contains a finite number of distinct u_i, we call them $(v_i)_{i \leq m}$. Then there is a permutation of the indexes such that the result of the permutation on β is $(v_0.v_1 \dots v_m)^\omega$. We know that ${v_m}^\omega \in f$ (see the lemma), and for all i, $f(v_i) = f$, so $v_0.v_1 \dots v_{m-1}.(v_m)^\omega \in f$. We call $\gamma = v_0.v_1 \dots v_{m-1}.(v_m)^\omega$. Because $\gamma \in f$, there is a sequence $(u'_i)_{i \in \mathbb{N}}$ such that $g'(u'_i) = \mathcal{B}^\omega$, $\gamma = u'_0.u'_1 \dots u'_i \dots$ and u'_i minimum. So there is a j such that $u'_0.u'_1 \dots u'_j = v_0.v_1 \dots v_{m-1}.{v_m}^n.w$ with

w prefix of v_m. Whatever i, $(u'_0.u'_1 \dots u'_i)^\omega \in f$, because $f = \Omega(g')$. So, by Proposition 1, $(v_0.v_1 \dots v_m)^\omega \in f$. This in turn means that $\beta \in f$.

If β contains infinitely many distinct u_i, we call them $(v_i)_{i \in \mathbb{N}}$. Necessarily, there is infinitely many 0's and 1's in β. So we have two v_i, w_0 and w_1 such that w_0 contains a 0 and w_1 contains a 1. As β is composed of 0 and 1, there is a permutation of the entries that transforms β in $(w_0.w_1)^\omega$. So we are back to the problem with β containing a finite number of u_i.

Thus, $\beta \in f$ whatever the case, which proves that $\alpha \in f$. We started from $\alpha \in \Omega(g)$, so $\Omega(g) \subset f$. Because we already proved $f \subset \Omega(g)$, we have $f = \Omega(g)$. □

7 BDGs with `rec` nodes: ω-deterministic Functions

We combine a finite behavior with infinite behaviors to express a wide variety of infinite functions. We call these functions ω-deterministic because we restrict the number of possible infinite behaviors at a given u to at most one iterative function.

Definition 6. *Let $f : \mathcal{B}^\omega \to \mathcal{B}$. f is ω-deterministic if and only if f is prefix regular and $\forall u$, $\Omega(\{v.\alpha \mid u.v^\omega \in f \text{ and } f(u.v) = f(u)\}) \subset f(u)$.*

7.1 The Decision Tree

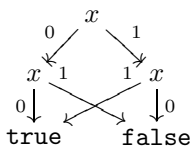
If the set $\{v \mid u.v^\omega \in f \text{ and } f(u.v) = f(u)\}$ is not empty, then it is possible, in the decision process, that we enter an infinite behavior. It must be signaled in the decision tree. To this end, we introduce a new kind of node in the decision tree, the `rec` node. The `rec` node signals that we must start a new infinite behavior, because before this node, we were in fact in the finite part of the function. The `rec` node has only one child. In the graphical representation, we will sometimes write $\overset{x}{\underset{t}{\downarrow}}$ for $\overset{x}{\underset{t}{\downarrow}}$ `rec` . After a `rec` node, we start the decision tree representing the iterative function. We know that when this decision tree comes to a `true`, we must start again just after the previous `rec` node. `false` nodes in the decision tree are replaced by the sequel of the description of the ω-deterministic function. As iterative functions of the ω-deterministic function are uniquely determined, and their representation is unique, the decision tree of the ω-deterministic function is unique.

Note that open functions are ω-deterministic. Their representation as an ω-deterministic function is the same as in the previous section, but with a `rec` preceding every `true`. It means also that the restriction of this representation to finite functions give the classical BDD, except for the `rec` preceding the `true`.

7.2 The Semantics of the Decision Tree

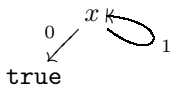
The semantics of the decision tree is defined in terms of a pseudo-decision process (it is not an actual decision process because it is infinite). The decision process reads a vector and uses a stack S and a current iterative tree, r. At the beginning, the stack is empty and r is the decision tree. When we come to a `true` node, we stack it and start again at r. When we come to a $\overset{\texttt{rec}}{\underset{t}{\downarrow}}$ node, r becomes t and we empty the stack. If we come to a `false` node, we stop the process. The process is a success if it doesn't stop and the stack is infinite.

Example 7.



when we read a 0, the current iterative tree becomes $\overset{x}{\underset{\texttt{true}}{\swarrow}\ \underset{\texttt{false}}{\searrow}}$. If we read a 1 after that, we stop on a failure, and if we read a 0, we stack a `true` and start again with the same iterative tree. So, after a 0, we can only have 0^ω. After a 1, the iterative tree becomes $\overset{x}{\underset{\texttt{false}}{\swarrow}\ \underset{\texttt{true}}{\searrow}}$, and this time, we can only have 1^ω. So this function is $\{0^\omega, 1^\omega\}$.

Example 8.



during the decision process, the iterative tree never changes. When we read a 0, we stack a `true` and start again. But each time we read a 1, we empty the stack. So the only vectors that stack an infinite number of `true` are the vectors ending by 0^ω.

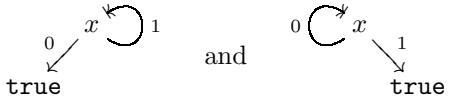
7.3 Boolean Operators

Theorem 5. *Let f and g be two ω-deterministic functions. Then $f \wedge g$ is ω-deterministic.*

The algorithm building the decision tree representing $f \wedge g$ identifies the loops, that is we come from a (t, u), which are subtrees of the decision trees representing f and g, and return to a (t, u). If in such a loop, we have not encountered any new `true` in any decision tree, we build a loop, if one decision process has progressed, we keep building the decision tree, and when both have been through a `true`, we add a `true` in the intersection.

ω-deterministic functions are not closed by union. As they are closed by intersection, it means that they are not closed by negation either.

Example 9 (Impossible Union). Let f_1 be the set of all vectors with a finite number of 1's, and f_2 the set of all vectors with a finite number of 0's. f_1 and f_2 are represented by:



Let $f = f_1 \vee f_2$. The set $\Omega\left(\{\, u.\alpha \mid u^\omega \in f \text{ and } f(u) = f\}\right)$ is the set of all vectors, but $(01)^\omega \notin f$, so f is not ω-deterministic.

7.4 Approximation

Theorem 6. *Whatever f prefix regular function, there is a least (for set inclusion) ω-deterministic function containing f.*

The process of building the best ω-deterministic function approximating a prefix regular function consists in adding the iterative functions defined by $\{v.\alpha |\, u.v^\omega \in f \text{ and } f(u.v) = f(u)\}$ to $f(u)$. This process starts at ε and keeps going for each `false` node in the representation of the iterative function, augmenting f at this point. The process is finite because f is prefix regular.

8 BDGs with Subscripts: Regular Functions

Regular functions are the closure of ω-deterministic functions by union. The idea is to allow a finite set of iterative functions to describe the infinite behavior at a given point in the function.

Definition 7. *Let $f : \mathcal{B}^\omega \to \mathcal{B}$. f is said to be* regular *if and only if f is prefix regular and there is a finite set of non-empty iterative functions $iter(f)$ such that $\forall \alpha \in f$, $\exists u$, $\exists g \in iter(f)$, $\alpha \in u.g$ and $g \subset f(u)$.*

Such functions are called regular because of the analogy with ω-regular sets of words of Büchi [6]. The only restriction imposed by the fact that we consider functions lies in the entry names, namely the entry names must be ultimately periodic to be finitely representable.

Theorem 7. *A function f is regular if and only if its entry names are ultimately periodic and the set of words in f is ω-regular in the sense of Büchi.*

The idea is that open functions define regular languages. If U is the regular language defined by an open function, then the associated iterative function is U^ω. The idea of the proof of the theorem is that an ω-regular language can be characterized as a finite union of $U.V^\omega$, with U and V regular languages.

Corollary 2. *If f and g are regular functions, then $f \wedge g$, $f \vee g$ and $\neg f$ are regular functions.*

It is an immediate consequence of the theorem, the closure properties of ω-regular languages, and the closure properties of the fact that the set of entry names is ultimately periodic.

Note that the set of regular functions is strictly smaller than the set of ω-regular languages:

Example 10. The set $\{0, 11\}^\omega$ is an ω-regular language, but not a regular function. Suppose two entries $i < j$ are equivalent. Then $0^{j-1}110^\omega$ is in the function, so $0^{i-1}10^{j-i}10^\omega$ is in the function too, because of the equivalence of entries i and j.

It is possible to define a decision tree that represents a regular function by subscripting some labels with a set of indexes corresponding to the iterative functions associated with the regular function. The meaning of this subscript is that we stack a `true` for each iterative function in the set of indexes. The problem is that this representation depends on the indexing of the iterative functions, and there is no canonical way of determining the set of iterative function that should be associated with a given regular function.

9 Conclusion

To achieve the representation of infinite functions, we presented a new insight on variable names which allows the sharing of some variable names. This sharing is compatible with every operation on classical BDDs, at no additional cost. It is even an improvement for classical BDDs, as it speeds up one of the basic operations on BDDs, the restriction operation.

We presented three classes of infinite functions which can be represented by extensions of BDDs. So far, the only extension that allowed the representation of infinite function was presented by Gupta and Fisher in [11] to allow inductive reasoning in circuit representation. Their extension corresponds to the first class (open functions), but without the uniqueness of the representation, because the loops have a name, which is arbitrary (and so there is no guarantee that the same loop encountered twice will be shared).

Our representation for open functions and ω-deterministic function have been tested in a prototype implementation in Java. Of course, this implementation cannot compete with the most involved ones on BDDs. It is, however, one of the advantages of using an extension of BDDs: many useful optimizations developed for BDDs could be useful, such as complement edges [3] or differential BDDs [1]. This is a direction for future work. Another direction for future work concerns the investigation over regular functions. These functions are closed by boolean operations, but we did not find a satisfactory unique representation with a decision tree yet. We believe the first two classes will already be quite useful. For example the first class (open function) is already an improvement over [11], and the second class (ω-deterministic) can express many useful properties of temporal logic [15]. This work is a step towards model checking and static analysis of the behavior of infinite systems, where properties depending on fairness can be expressed and manipulated efficiently using BDGs.

Acknowledgments: Many thanks to Patrick Cousot for reading this work carefully and for many useful comments. Ian Mackie corrected many English mistakes in the paper.

References

[1] Anuchitanakul, A., Manna, Z., and Uribe, T. E. Differential BDDs. In *Computer Science Today* (September 1995), J. van Leeuwen, Ed., vol. 1000 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 218–233.

[2] Bagnara, R. A reactive implementation of pos using ROBDDs. In *8th International Symposium on Programming Languages, Implementation, Logic and Programs* (September 1996), H. Kuchen and S. D. Swierstra, Eds., vol. 1140 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 107–121.

[3] Billon, J. P. Perfect normal form for discrete programs. Tech. Rep. 87039, BULL, 1987.

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

[5] Bryant, R. E. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys 24*, 3 (September 1992), 293–318.

[6] Büchi, J. R. On a decision method in restricted second order arithmetics. In *International Congress on Logic, Methodology and Philosophy of Science* (1960), E. Nagel et al., Eds., Stanford University Press.

[7] Burch, J. R., Clarke, E. M., McMillan, K. L., Dill, D. L., and Hwang, J. Symbolic modek checking: 10^{20} states and beyond. In *Fifth Annual Symposium on Logic in Computer Science* (June 1990).

[8] Corsini, M.-M., Musumbi, K., and Rauzy, A. The μ-calculus over finite domains as an abstract semantics of Prolog. In *Workshop on Static Analysis* (September 1992), M. Billaud, P. Castran, M.-M. Corsini, K. Musumbu, and A. Rauzy, Eds., no. 81–82 in Bigre.

[9] Cousot, P., and Cousot, R. Abstract interpretation; a unified lattice model for static analysis of programs by construction of approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages (POPL '77)* (1977), pp. 238–252.

[10] Fujita, M., Fujisawa, H., and Kawato, N. Evaluation and improvements of boolean comparison method based on binary decision diagram. In *International Conference on Computer-Aided Design* (November 1988), IEEE, pp. 3–5.

[11] Gupta, A., and Fisher, A. L. Parametric circuit representation using inductive boolean functions. In *Computer Aided Verification* (1993), C. Courcoubetis, Ed., vol. 697 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 15–28.

[12] Jeong, S.-W., Plessier, B., Hachtel, G. D., and Somenzi, F. Variable ordering and selection for FSM traversal. In *International Conference on Computer-Aided Design* (November 1991), IEEE, pp. 476–479.

[13] Le Charlier, B., and van Hentenryck, P. Groundness analysis for Prolog: Implementation and evaluation of the domain prop. In *PEPM'93* (1993).

[14] Madre, J.-C., and Coudert, O. A logically complete reasoning maintenance system based on a logical constraint solver. In *12th International Joint Conference on Artificial Intelligence* (1991), pp. 294–299.

[15] Manna, Z., and Pnueli, A. The anchored version of the temporal framework. In *Linear Time, Branching Time, and Partial Order in Logics and Models for Concurrency* (May/June 1988), J. W. de Bakker, W. P. de Roever, and G. Rozenberg, Eds., vol. 354 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 201–284.

[16] Mauborgne, L. Abstract interpretation using typed decision graphs. *Science of Computer Programming 31*, 1 (May 1998), 91–112.

[17] Mauborgne, L. *Representation of Sets of Trees for Abstract Interpretation.* PhD thesis, École Polytechnique, 1999.

[18] Michie, D. "memo" functions and machine learning. *Nature 218* (April 1968), 19–22.