UNIVERSITÀ DEGLI STUDI DI PARMA
Facoltà di Scienze Matematiche Fisiche e Naturali

Corso di Laurea in Informatica
Tesi di Laurea Triennale

CORAL: a modern C++ library for the manipulation of Boolean functions

Candidato Fabio Bossi
Relatore Roberto Bagnara

Anno Accademico 2007/2008

A mio padre

Abstract

The purpose of this work is to describe the CORAL library, which provides an efficient implementation of Boolean functions; it has been known for a long time that almost all of the interesting computations on Boolean functions, such as the satisfiability problem, are NP-hard, so this is not an easy task. The library was originally designed to be used by the China analyzer, in particular in the context of the *groundness analysis* problem of CLP languages, for which high performance operations on very large Boolean functions are needed (the internal representation of the functions involved can reach a size in the order of megabytes). However, we are confident that the library can and will be used succesfully in other areas of application, and we will not discuss particular applications in this work. In order to perform operations on Boolean functions we need a way to represent them: CORAL utilizes *reduced ordered binary decision diagrams* (ROBDDs), which means that the most important part of this work will deal with their properties and the algorithms used to manipulate them. We also describe how CORAL supports alternative, mixed representations to optimize efficiency and memory occupation in contexts where *entailed, disentailed* and/or *equivalent variables* are frequent.

Contents

Organization

Chapter 1. Introduces the fundamental concepts for Boolean functions.

Chapter 2. Introduces ROBDDs, the fundamental data structure used by CORAL to represent and manipulate Boolean expressions, discusses their properties and shows an example of how they can be used to implement operators on Boolean functions.

Chapter 3. Describes CORAL's fundamental algorithms in detail.

Chapter 4. Discusses how the library supports different internal representations of Boolean expressions in a way that is reasonably transparent to the end user by applying *policy-based class design*.

Chapter 5. Provides the results of experimental evaluation of CORAL's performance.

Chapter 6. Briefly draws conclusions.

1 Boolean functions

1.1 Fundamental concepts

1.1.1 Definition of Boolean functions

Boolean functions are functions that depend on a set of Boolean variables. The variables and the function itself have their values in $\{0, 1\}$. We will also refer to the elements 0 and 1 as false and true respectively. Formally, we denote the infinite and numerable set of *Boolean variables* as Vars. We also define a total order relation $\prec$ over Vars that allows us to assign an index $i \in \mathbb{N}$ to each variable $x_i \in \text{Vars}$. We can now give the formal definition of *Boolean valuation* and *Boolean function*.

Definition 1. (Boolean valuations and Boolean functions.) The set of *Boolean valuations* over Vars is defined as $\mathcal{A} \stackrel{\text{def}}{=} \text{Vars} \to \{0, 1\}$. The set of Boolean functions is defined as $\mathcal{F} \stackrel{\text{def}}{=} \mathcal{A} \to \{0, 1\}$.

Sometimes we may use *variable assignment* as a synonym for Boolean valuation.

1.1.2 Denoting specific Boolean functions

We will use a simple notation to refer to specific Boolean functions, exploiting the fact that they can all be obtained by applying a few simple operators on a few elementary functions. We can start by defining the Boolean functions $\bot$ and $\top$ as the Boolean functions which evaluate to false and true respectively in any given Boolean valuation. For each variable $x \in \text{Vars}$, we also write x to denote the Boolean function f such that $\forall a \in \mathcal{A} : f(a) = 1 \iff a(x) = 1$. It should always be clear whether we are referring to the variable or to the function in a specific context.

Now we define a few basic binary operations over $\mathcal{F}$. For each pair of Boolean functions $f_1, f_2 \in \mathcal{F}$, we write $f_1 \wedge f_2$ to denote the Boolean function $g \in \mathcal{F}$ such that for each $a \in \mathcal{A}$, $g(a) = 1 \iff (f_1(a) = 1 \wedge f_2(a) = 1)$. We define all other common operations analogously; here is a complete list:

Definition 2. (Binary operators on Boolean functions.) $\forall f_1, f_2 \in \mathcal{F}$:

$$f_1 \wedge f_2 \stackrel{\text{def}}{=} g \in \mathcal{F} \,.\, \forall a \in \mathcal{A} : g(a) = 1 \iff \big(f_1(a) = 1 \wedge f_2(a) = 1\big),$$

$$f_1 \vee f_2 \stackrel{\text{def}}{=} g \in \mathcal{F} \,.\, \forall a \in \mathcal{A} : g(a) = 1 \iff \big(f_1(a) = 1 \vee f_2(a) = 1\big),$$

$$f_1 \oplus f_2 \stackrel{\text{def}}{=} g \in \mathcal{F} \,.\, \forall a \in \mathcal{A} : g(a) = 1 \iff \big(f_1(a) \neq f_2(a)\big),$$

$$f_1 \Leftrightarrow f_2 \stackrel{\text{def}}{=} g \in \mathcal{F} \,.\, \forall a \in \mathcal{A} : g(a) = 1 \iff \big(f_1(a) = f_2(a)\big),$$

$$f_1 \Rightarrow f_2 \stackrel{\text{def}}{=} g \in \mathcal{F} \,.\, \forall a \in \mathcal{A} : g(a) = 1 \iff \big(f_1(a) = 0 \vee (f_1(a) = 1 \wedge f_2(a) = 1)\big).$$

The $\wedge$ operator has precedence over the $\vee$ operator.

For each $f \in \mathcal{F}$, we define $\overline{f}$ (sometimes denoted as $\neg f$) as the Boolean function such that $\forall a \in \mathcal{A} : \overline{f}(a) = 1 \iff f(a) = 0$. For each $i, t, e \in \mathcal{F}$ we also define $if\ i\ then\ t\ else\ e \stackrel{\text{def}}{=} \big(i \wedge t\big) \vee \big(\overline{i} \wedge e\big)$.

Finally, we need a notation for Boolean functions obtained by forcing some variables in another Boolean function to a given truth value. For each valuation $a \in \mathcal{A}$, each $x, y \in \text{Vars}$, and each $c \in \{0, 1\}$, the valuation $a[c/x] \in \mathcal{A}$ is defined as

$$a[c/x](y) \stackrel{\text{def}}{=} \begin{cases} c, & \text{if } x = y; \\ a(y), & \text{otherwise.} \end{cases}$$

We extend this to a set of variables by defining, for each $X = \{v_1, v_2, \dots v_n\} \subseteq \text{Vars}$, the Boolean valuation $a[c/X] \stackrel{\text{def}}{=} a[c/v_1][c/v_2] \dots [c/v_n]$. Now we can define, for each $f \in \mathcal{F}$, each $x \in \text{Vars}$, each $a \in \mathcal{A}$ and each $c \in \{0, 1\}$, the Boolean function $f[c/x] \in \mathcal{F}$ (also denoted as $f\ |_{x \leftarrow c}$) as $f[c/x](a) \stackrel{\text{def}}{=} f(a[c/x])$. Like we did with Boolean valuations, we define for each $X = \{v_1, v_2, \dots v_n\} \subseteq \text{Vars}$ the Boolean function $f[c/X] \stackrel{\text{def}}{=} f[c/v_1][c/v_2] \dots [c/v_n]$. We also define for each $f \in \mathcal{F}$, each $x, y \in \text{Vars}$, and each $a \in \mathcal{A}$, the Boolean function $f[x/y] \in \mathcal{F}$ as $f[x/y](a) \stackrel{\text{def}}{=} f(a[a(x)/y])$.

1.1.3 Notable sets of variables

We now introduce, for any given Boolean function f, the set of variables on which f depends, the sets of its *entailed* and *disentailed* variables, and the equivalence relation representing its *equivalent* variables; calculating them is necessary for many applications.

Definition 3. (Variable dependency. Entailed, disentailed and equivalent variables.) For each $f \in \mathcal{F}$, the set of variables on which f depends, the set of entailed variables for f, the set of disentailed variables for f, and the equivalence relation representing the equivalent variables for f are given, respectively, by:

$$\begin{aligned}
\text{vars}(f) &\stackrel{\text{def}}{=} \big\{\, x \in \text{Vars} \;\big|\; \exists a \in \mathcal{A}\ .\ f\big(a[0/x]\big) \neq f\big(a[1/x]\big) \,\big\}, \\
\text{true}(f) &\stackrel{\text{def}}{=} \big\{\, x \in \text{Vars} \;\big|\; \forall a \in \mathcal{A} : f(a) = 1 \implies a(x) = 1 \,\big\}, \\
\text{false}(f) &\stackrel{\text{def}}{=} \big\{\, x \in \text{Vars} \;\big|\; \forall a \in \mathcal{A} : f(a) = 1 \implies a(x) = 0 \,\big\}, \\
\text{equiv}(f) &\stackrel{\text{def}}{=} \big\{\, (x, y) \in \text{Vars}^2 \;\big|\; \forall a \in \mathcal{A} : f(a) = 1 \implies a(x) = a(y) \,\big\}.
\end{aligned}$$

1.1.4 Equivalence relations over sets of variables

Let $\mathcal{L}$ be the set of all equivalence relations over a finite set of variables and consider a particular equivalence relation $L \in \mathcal{L}$. We define the leader of an equivalence class in L as the minimum variable in the equivalence class; we also say that variable v_l is the leader of variable v_i for L when v_l is the leader of the equivalence class in L that contains v_i (note that each leader is the leader of itself), and define $\lambda_L(v_i) \stackrel{\text{def}}{=} v_l$. The set of all leaders for

L is denoted as leaders(L) and its domain is denoted as dom(L). If $E_1, E_2 \in \mathcal{L}$ we define $E_1 \vee E_2$ as the intersection of E_1 and E_2 and $E_1 \wedge E_2$ as the transitive closure of the union of E_1 and E_2. Formally it is defined inductively, for each $i \in \mathbb{N},\ i > 0$, by:

$$
\begin{aligned}
&E_1 \wedge_0 E_2 \stackrel{\text{def}}{=} E_1 \cup E_2,\\
&E_1 \wedge_i E_2 \stackrel{\text{def}}{=} \big\{(x,z) \in \mathrm{Vars}^2 \;\big|\; \exists y \in \mathrm{Vars}\,.\,\big(\exists(x,y),(y,z) \in (E_1 \wedge_{i-1} E_2)\big)\big\},\\
&E_1 \wedge E_2 \stackrel{\text{def}}{=} \bigcup_{i\in\mathbb{N}} (E_1 \wedge_i E_2).
\end{aligned}
$$

Finally, we define the domain of an equivalence relation $L \in \mathcal{L}$ as

$$\mathrm{dom}(L) \stackrel{\text{def}}{=} \big\{x \in \mathrm{Vars}\,.\,\exists(x,y) \in L\big\}.$$

1.1.5 Projections

Given a Boolean function $f \in \mathcal{F}$ and a variable $x \in \mathrm{Vars}$, we define the Boolean function

$$\exists_x f \stackrel{\text{def}}{=} f[0/x] \vee f[1/x]. \tag{1.1}$$

This equation is also known as *Schröder's elimination principle* and is often exploited by the algorithms that compute projections. Given a finite set of variables $\{v_1, v_2, \ldots v_n\} \subseteq \mathrm{Vars}$ we also define the Boolean function

$$\exists_{\{v_1,v_2,\ldots v_n\}} f \stackrel{\text{def}}{=} \exists_{v_1} f \wedge \exists_{v_2} f \wedge \ldots \exists_{v_n} f.$$

We finally define the *projection* of a Boolean function f onto a finite set of variables X as the Boolean function

$$f\,|_X \stackrel{\text{def}}{=} \exists_{\{\mathrm{Vars}\setminus X\}} f.$$

1.1.6 Positive Boolean functions

We define the set of positive Boolean functions *Pos* as the set of all Boolean functions that evaluate to true in the variable assignment which assigns 1 to all variables in Vars.

Definition 4. (Positive Boolean functions.) If $t \in \mathcal{A}$ is the Boolean valuation such that $\forall v \in \mathrm{Vars} : t(v) = 1$, we define

$$Pos \stackrel{\text{def}}{=} \big\{\, f \in \mathcal{F} \;\big|\; f(t) = 1 \,\big\}$$

Marriott and Søndergaard (see [MS93]) have found this class of Boolean functions to be a good abstract domain for the problem of groundness analysis: that is why CORAL provides some specialized algorithms for it.

1.1.7 Shannon expansion

Theorem 1. (Shannon expansion.) *Let $f \in \mathcal{F}$ and $x \in Vars$. Then:*

$$f = (\overline{x} \wedge f[0/x]) \vee (x \wedge f[1/x]).$$

Proof. Trivial: simply take a generic variable assignment $a \in \mathcal{A}$ and evaluate the function in it for both cases $a(x) = 0$ and $a(x) = 1$. □

This is very important for us because operators on Boolean functions commute with the Shannon expansion. For example, if $\bullet$ is a binary operator, then

$$f \bullet g = (\overline{x} \wedge (f[0/x] \bullet g[0/x])) \vee (x \wedge (f[1/x] \bullet g[1/x])). \tag{1.2}$$

This equation can be easily extended to non-binary operators.

2 Reduced Ordered Binary Decision Diagrams

2.1 Definition

2.1.1 ROBDD definition

Reduced Ordered Binary Decision Diagrams (ROBDDs), first introduced by Randal E. Bryant in [Bry86] along with the basic algorithms that manipulate them, are the fundamental data structure used by CORAL to represent Boolean functions. A ROBDD is a rooted, directed, and acyclic graph where leaf nodes are either **0** or **1** (the *terminal nodes*) and each non-leaf node n is labeled with a Boolean variable $n_{\text{var}} \in \text{Vars}$ on which the represented Boolean function depends ($n_{\text{var}} \in \text{vars}(f)$). Each non-leaf node has two successors n_{false} and n_{true}, called the *false successor* and the *true successor* respectively. Both successors can be either a leaf node or a non-leaf node labeled with a variable that is greater than n_{var} (this explains the *Ordered* part of the name). Finally, no duplicate nodes must be present (this explains the *Reduced* part of the name). As Bryant himself showed, it is possible to transform any Ordered Binary Decision Diagram into a ROBDD by applying a few simple reduction rules:

Remove duplicate terminal nodes: Keep no more than one copy of the terminal nodes **0** and **1** and redirect all edges that were directed towards the removed duplicates to the remaining copies.

Remove duplicate non-terminal nodes: If two or more non-terminal nodes are labeled with the same variable and share the same true and false successors, keep only one copy and redirect all edges that were directed towards the removed duplicates to the remaining copy.

Remove nodes having a unique successor: Remove all nodes that have their true successor equal to their false successor and redirect all edges that were directed towards them to their successor.

The rules must be applied repeatedly until a fixpoint is reached, as applying one rule may lead to further reduction possibilities for the others. We now give a formal definition for ROBDDs:

Definition 5. (ROBDDs.) If N is the set of nodes of a ROBDD then N satisfies

$$N \subseteq \{\mathbf{0}, \mathbf{1}\} \cup \text{Vars} \times N \times N.$$

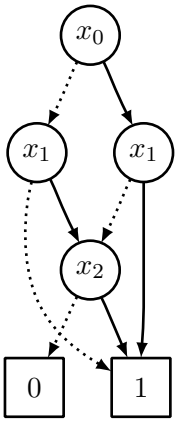

Figure 2.1: A ROBDD for Boolean function $(x_0 \Leftrightarrow x_1) \vee x_2$.

The nodes **0** and **1** are called *terminal nodes*. All the other nodes in N are called *non-terminal* nodes. For each non-terminal node $n \in N$, $n = (x, f, t)$, $n_{\text{var}} \in \text{Vars}$ denotes x, that is the variable associated with n, $n_{\text{false}} \in N$ denotes f, that is the false successor of n, and $n_{\text{true}} \in N$ denotes t, that is the true successor of n. With this notation, N must also satisfy the irredundancy and the ordering conditions: for each non-terminal nodes $n, m \in N$ $n_{\text{false}} \neq n_{\text{true}}$ and $\big(m = n_{\text{false}} \text{ or } m = n_{\text{true}}\big) \implies \big(m \in \{\mathbf{0}, \mathbf{1}\} \text{ or } n_{\text{var}} \prec m_{\text{var}}\big)$. Moreover, N is rooted and connected, which means that there exists $r \in N$ (the *root*) such that

$$\forall n \in N \setminus \{r\} : \exists m \in N \,.\, \big(n = m_{\mathbf{0}} \text{ or } n = m_{\mathbf{1}}\big).$$

A ROBDD is a pair (r, N) that satisfies the above conditions. The set of all ROBDDs is denoted by $\mathcal{D}$. The set of all ROBDD nodes is denoted by $\mathcal{N}$. For each ROBDD (r, N) we also define $N_{false} \subset N$ as the set of the nodes of the ROBDD rooted at r_{false} and $N_{true} \subset N$ as the set of the nodes of the ROBDD rooted at r_{true}.

For simplicity, we will sometimes confuse a ROBDD with its root note. In particular, we may use **0** and **1** to refer to the ROBDDs $(\mathbf{0}, \{\mathbf{0}\})$ and $(\mathbf{1}, \{\mathbf{1}\})$ respectively.

We will represent ROBDDs as graphs in figures, using a dotted line for the edges directed from a node to its false successor (the *false edges*) and a full line for the edges directed from a node to its true successor (the *true edges*). Non-terminal nodes will be drawn as circles while terminal nodes will be drawn as squares. An example is given in Figure 2.1.

2.1.2 ROBDD semantics

The semantics of a ROBDD is very simple: each ROBDD represents a Boolean function $f \in \mathcal{F}$. For each variable assignment $a \in \mathcal{A}$, $f(a) \in \{0, 1\}$ is obtained by traversing the graph starting from the root: whenever we encounter a non-terminal node n, we move

onto n_{false} if $a(n_{\text{var}}) = 0$, otherwise $a(n_{\text{var}}) = 1$ and we move onto n_{true}. If we reach the **0** node then $f(a) = 0$, otherwise we reach the **1** node and $f(a) = 1$. The semantics, according to [BS98], can be formally defined by a function which maps each ROBDD into the represented Boolean function:

Definition 6. (Semantics of ROBDDs.) The function $[\![\cdot]\!]_{\mathcal{D}} \colon \mathcal{D} \to \mathcal{F}$ is given, for each $(r, N) \in \mathcal{D}$, by

$$[\![(r,N)]\!]_{\mathcal{D}} \stackrel{\text{def}}{=} \begin{cases} \bot, & \text{if } r = \mathbf{0}; \\ \top, & \text{if } r = \mathbf{1}; \\ \left(r_{\text{var}} \wedge [\![(r_{\text{true}}, N_{true})]\!]_{\mathcal{D}}\right) \vee \left(\neg r_{\text{var}} \wedge [\![(r_{\text{false}}, N_{false})]\!]_{\mathcal{D}}\right), & \text{otherwise.} \end{cases}$$

For the sake of simplicity, we will sometimes write $[\![(r)]\!]_{\mathcal{D}}$ instead of $[\![(r, N)]\!]_{\mathcal{D}}$.

2.2 ROBDD properties

2.2.1 Canonicity

Theorem 2. (Canonicity of ROBDDs.) *Given a fixed variable ordering $\prec$, we have that for each $f \in \mathcal{F}$ there exists one and only one ROBDD $(r, N) \in \mathcal{D}$ such that $[\![(r, N)]\!]_{\mathcal{D}} = f$.*

Proof. See [Bry86]. □

This has many important consequences for the CORAL implementation: for example, we have that two Boolean functions are identical if and only if they are represented by the same ROBDD. CORAL keeps only one copy of each ROBDD into memory, so that equality test on ROBDDs can be performed simply by comparing their memory addresses.

2.2.2 Influence of variable ordering

The number of nodes of ROBDDs is extremely critical for the efficiency of the algorithms used to manipulate them, and the size of the ROBDD which represents a particular Boolean function may vary depending on the chosen variable ordering. Unfortunately, for any fixed variable ordering, it is possible to find some Boolean functions such that their representative ROBDD's size is exponential in the number of variables. For example, Bryant has verified in [Bry92] that while ROBDDs for symmetric Boolean functions may have sizes that range from being linear to quadratic in the number of variables depending on the chosen ordering, yet ROBDDs for functions representing integer addition may range from linear to exponential; for integer multiplication we even have exponentiality for all variable orderings. This is why some ROBDD implementations have built-in mechanisms for changing variable ordering dynamically, relying for example on heuristical analysis. CORAL instead uses fixed variablable ordering and relies on other mechanisms (which will be discussed later) for the purpose of keeping the ROBDDs as small as possible. In Figure 2.2 we give an example of two ROBDDs representing the same Boolean function $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$ using two different variable orderings.

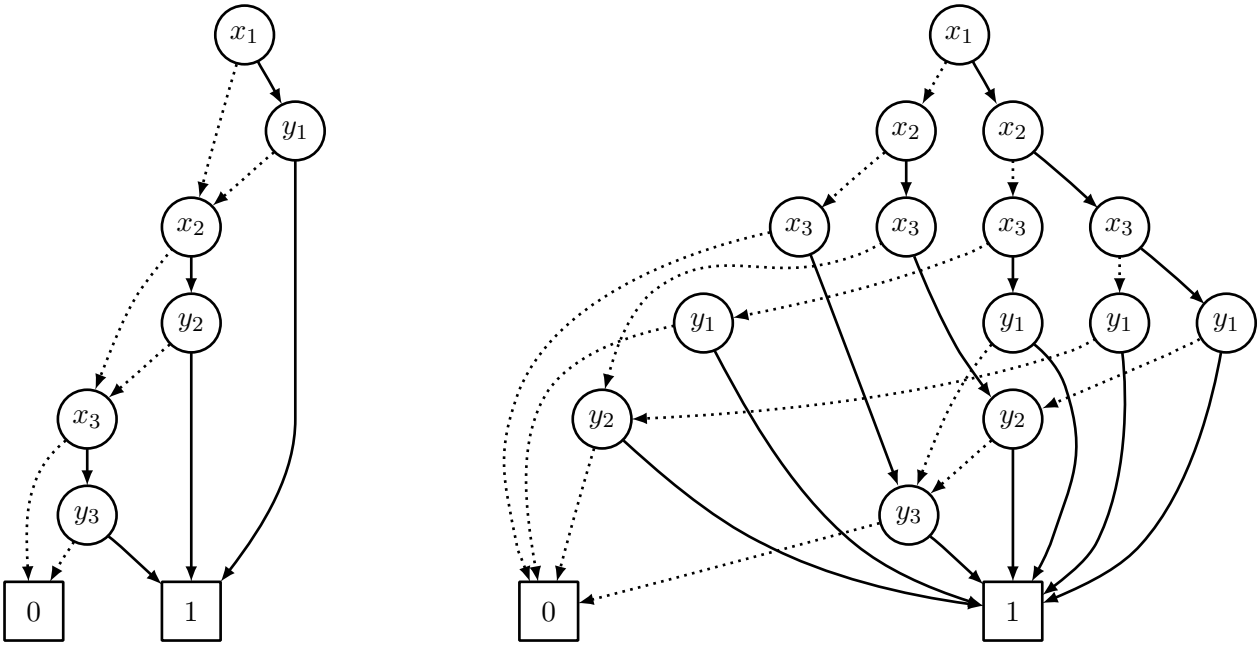

Figure 2.2: Two ROBDDs representing the Boolean function $(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$ using different variable orderings.

2.3 Operations on ROBDDs

2.3.1 Implementation of algebraic operators

The basic idea

We will now show how it is possible to use ROBDDs for implementing operations on Boolean functions, taking the application of a generic binary operator as an example. The fundamental trick consists in exploiting Equation 1.2, combined with the fact that for each ROBDD (r, N) representing the Boolean function $f = [\![(r)]\!]_{\mathcal{D}}$, for each variable v such that $v \preceq r_{\text{var}}$ and for each $b \in \{0, 1\}$ we have:

$$f[b/v] = \begin{cases} f, & \text{if } v \prec r_{\text{var}} \\ [\![(r_{\text{false}})]\!]_{\mathcal{D}}, & \text{if } v = r_{\text{var}} \text{ and } b = 0 \\ [\![(r_{\text{true}})]\!]_{\mathcal{D}}, & \text{if } v = r_{\text{var}} \text{ and } b = 1 \end{cases} \tag{2.1}$$

Given these results, we can implement a binary operator $\bullet$ with a recursive procedure starting from the roots of the two arguments, using an algorithm that has been presented in [Bry86] and improved in [Bry92]. Each recursive call considers one node for each of the two ROBDD arguments and generates one node of the resulting ROBDD. The base case is encountered when both nodes are terminal nodes: the result is a terminal node obtained by applying $\bullet$ on the truth values represented by the two terminal nodes. If the

operator has a dominant value (for example 0 for $\wedge$ and 1 for $\vee$), we can define another base case when one of the two nodes is the terminal node corresponding to the dominant value and the other is a generic node (terminal or non-terminal): the result is the terminal node corresponding to the dominant value.

For the non-base case we are considering two nodes n^1 and n^2, where n^1 is a node of the first argument and n^2 is a node of the second argument. We choose $v = \min(n^1_{\text{var}}, n^2_{\text{var}})$ (or, if one of the two nodes is terminal, the variable by which the other is labeled) as the *splitting variable*: we can now apply Shannon expansion with respect to variable v to the Boolean functions $f = [\![n^1]\!]_{\mathcal{D}}$ and $g = [\![n^2]\!]_{\mathcal{D}}$. Equation 2.1 allows us to compute $f[0/v]$, $f[1/v]$, $g[0/v]$ and $g[1/v]$: the first two will be represented by two nodes of the first argument, while the last two will be represented by two nodes of the second argument; if one of the two nodes n^1 or n^2 was not labeled with the splitting variable, then two of the four computed nodes will of course be, in fact, the same node and will coincide with n^1 or n^2.

The result of the non-base case will be, according to Shannon expansion, a non-terminal node labeled with v; its true successor is obtained by calling the algorithm recursively on the computed nodes representing $f[1/v]$ and $g[1/v]$, while its false successor is obtained by calling the algorithm recursively on the computed nodes representing $f[0/v]$ and $g[0/v]$. This algorithm, without further optimization, produces a correct unreduced graph, but we can do much better while building a reduced graph by adding a few tweaks.

Efficiency tweaks and hash tables

In order to obtain a reduced graph, we apply reduction rules to the result at the end of each recursive call: if the result of the recursive call has its true successor equal to its false successor, we discard the node and connect its predecessor directly to its successor.

In order to avoid duplicate nodes, we keep an hash table with keys in $\text{Vars} \times \mathcal{N} \times \mathcal{N}$: each key refers to a node and contains the variable by which it is labeled, a pointer to its true successor and a pointer to its false successor (of course the two pointers must be different in order to comply with reduction rules), while the value of an entry contains a pointer to the represented node if it is already present in memory or contains NULL otherwise. After each recursive call we look for the resulting node into memory by performing an hash lookup: if the node is found then it is returned as the result, otherwise the new node is created and an hash entry corresponding to it is added. CORAL in fact utilizes a different kind of hash table, as the table would actually grow too large for our applications if we kept an entry for each allocated node. We will not describe the hash function here; just be aware that the value of an entry is not a single pointer but a list of pointers, so a lookup may require non-constant time in the worst case.

For each binary operator, we also keep another hash table that works as a cache for the purpose of avoiding multiple recursive calls on the same pairs of nodes: the table has keys in $\mathcal{N} \times \mathcal{N}$ and values in $\mathcal{N}$, where each value contains the result of the the application of the binary operator on the pair of nodes represented by the key. At the beginning of each recursive call we check if the resulting node has already been computed by performing an hash lookup: if the result is found then it is returned immediately, otherwise it is

computed normally and an hash entry corresponding to the performed computation is added. Since we perform at most one recursive call for each pair of nodes and provided that our implementation of hash tables is good enough to bring the average cost of each recursive valuation to a constant value, the worst case complexity of the algorithm is in the order of nm, where n and m are the number of nodes in the first and the second ROBDD respectively.

The implementation of operators with more than two arguments can be easily extended from this one.

2.3.2 Implementation of the restriction operation

Given a ROBDD representing the Boolean function f, $v \in \text{Vars}$ and $b \in \{0, 1\}$, we can compute $f[b/x]$ with a recursive procedure starting from the root of the ROBDD: each recursive call considers one node of the argument and returns one node of the resulting ROBDD. The base case is when the considered node is labeled with v: here we must simply return its true successor if $b = 1$ or its false successor if $b = 0$. For the recursive case we simply make recursive calls on both successors. The reduction of the result and the avoidance of multiple recursive calls can be guaranteed by using the usual techniques, with the obvious exception that the keys of the hash table that is used to avoid multiple recursive calls are composed by a single pointer.

The techniques described in this chapter can be used to implement most operations on Boolean functions, although CORAL utilizes a few other tweaks which will be presented in the next two chapters.

3 Algorithms in CORAL

3.1 Basic data structures

3.1.1 Sets of Boolean variables

The type of variable indexes is `index_type`. A single Boolean variable in CORAL is represented by an object of class `Variable`; it is a very simple class whose objects can be built using a constructor that takes an `index_type` which represents the index of the variable. To represent (finite) sets of variables we use the `Bit_Set` class, which in turn represents a set using a bitmap as its private data member: a specific variable v_i is contained in the set if and only if the bit in the position i of the bitmap exists and is set to 1. The type of this bitmap is `mpz_t`, which is provided by the GMP library; using bitmaps allows us to implement the common operations on sets such as union, intersection, difference and inclusion tests efficiently. The end user in fact utilizes class `Variables_Set` instead, which is a simple wrapper for `Bit_Set` that is a bit closer to the concept of variable.

3.1.2 Equivalence relations

An object of class `Equivalence_Relation` represents an equivalence relation $L \in \mathcal{L}$. The internal representation consists of a vector of elements of type `index_type`. Each element of the vector in position i is set to value l if v_l is the leader of v_i for the relation, or to the special value `NOT_IN_DOMAIN` if $v_i \notin \mathrm{dom}(L)$. CORAL provides some functions to perform basic computations on Boolean function such as the intersection $\vee$ and the transitive closure of the union $\wedge$.

3.1.3 Hash table

An hash table `hash_table` is used to ensure that there are no duplicate ROBDD nodes, as described in Subsection 2.3.1. Its method `find_or_insert` performs a lookup taking the three components of the key (the variable by which the root of the ROBDD is labeled, a pointer to the true successor and a pointer to the false successor) as its arguments: if an entry is found then the the return value is a pointer to the ROBDD that has already been constructed, otherwise a new node is allocated, a cache entry for it is added and the return value is a pointer to this new node. This method also handles the particular case where the two pointer arguments are in fact the same pointer simply by returning this pointer, in order to respect the third reduction rule. The static method `make_node` of class ROBDD is simply a wrapper for `find_or_insert` that takes the same arguments and returns the same pointer. From now on, whenever we are describing a ROBDD

algorithm and write that a node is created, we will leave the fact that we are calling `make_node` (thus applying the third reduction rule) implicit.

3.1.4 Caches

CORAL keeps associative caches for most of its ROBDD operations. We will not give a list here as their names are easily identifiable by the `Cache` suffix in the source code. For simplicity we will not mention them in the pseudo-codes.

3.2 ROBDD algorithms

3.2.1 Memory management

CORAL's class `ROBDD` does not distinguish a ROBDD from its root node and uses its own customized mechanisms for the allocation of ROBDD nodes. `ROBDD` is derived from the `Reference_Counted_Object` class in order for ROBDDs to be reference counted, which means that each ROBDD is normally deallocated (by a garbage collection algorithm) only after there are no more references to it. The end user, in fact, does not use `ROBDD` directly: instead he utilizes the `ROBDD_Ptr` class, which is nothing more than a wrapper for a ROBDD pointer that also automatically adjusts reference counters of ROBDDs whenever the underlying pointer is changed.

All ROBDD nodes allocated by CORAL are kept inside an array with variable length, which is formed by a certain number of memory chunks. CORAL counts the number of allocated chunks and will throw an exception of type `Too_Many_BDD_Nodes` if it exceeds `MAX_CHUNKS_BEFORE_EXCEPTION` and the public Boolean static member `ROBDD::dont_throw` is set to false, which means that setting this latter value to false allows the end user to be notified when the memory occupation of ROBDDs is becoming too large. In this case, the normal behavior consists of calling `ROBDD::emergency_cleanup()` after catching the expression in order to regain some memory.

3.2.2 Algorithms with upward approximation

Algorithms with upward approximation, characterized by the substring `up` in their names, can be used on large ROBDDs in order to obtain more efficiency at the expense of some precision in the result. Each Boolean function g such that $f \Longrightarrow g$ is an upward approximation of function f. These algorithms return an upward approximation of the result by trying to perform their corresponding non-approximated algorithm and checking if it throws `Too_Many_BDD_Nodes`: if the exception is not thrown then the exact result is returned, otherwise we return an upward approximation for it. For example the `and_up` operator, which is the upward approximation version of the conjunction operator `and_ite`, approximates $f \wedge g$ with f when `Too_Many_BDD_Nodes` is thrown; its implementation is shown in Listing 3.1.

```
inline const ROBDD*
ROBDD::and_up(const ROBDD* y) const {
  try {
    return and_cache.find_or_compute(this, y);
  }
  catch(Too_Many_BDD_Nodes) {
    ROBDD::emergency_cleanup();
    // x is a sound approximation of x & y.
    return this;
  }
}
```

Listing 3.1: Implementation of the conjunction operator with upward approximation

3.2.3 Complexity limitation

The static members of class ROBDD `equivalent_countdown` and `rename_countdown` are unsigned integers used to provide a complexity upper bound for some operations on ROBDDs: the first is used for operations that look for equivalent variables and the second for operations that perform variable renaming. These operations automatically decrement the counters; when they reach `0` an exception of type `Equivalent_Too_Costly` or `Rename_Too_Costly` is thrown.

3.2.4 Conjunction and disjunction

These algorithms, named `and_ite` and `or_ite`, are nothing more than a particular case of the application of a binary operator presented in subsection 2.3.1 and differ only for their base cases. For conjunction, the base cases are when one of the two ROBDDs is the **1** or the **0** node. In the first case the result is the other ROBDD, while in the second case the result is **0**. For disjunction, the base cases are the same, but we return **1** in the first case and the other ROBDD in the second case. Pseudo-code for conjunction and disjunction is shown in Algorithm 1 and Algorithm 2.

3.2.5 if/then/else operator

The if/then/else operator `ite` is an example of ternary operator: given three ROBDDs i, t, e representing the Boolean functions i, t and e respectively (note that we deliberately confuse ROBDDs with the represented functions), it computes the ROBDD representing the Boolean function $if\ i\ then\ t\ else\ e$.

Require: two ROBDD nodes n and m
  function and_ite(n, m)
  if $n = \mathbf{1}$ **then**
    return m
  else if $m = \mathbf{1}$ **then**
    return n
  else if $n = \mathbf{0}$ **or** $m = \mathbf{0}$ **then**
    return 0
  else if $n_{\text{var}} \prec m_{\text{var}}$ **then**
    return make_node$\big(n_{\text{var}}, \text{and_ite}(n_{\text{true}}, m), \text{and_ite}(n_{\text{false}}, m)\big)$
  else if $m_{\text{var}} \prec n_{\text{var}}$ **then**
    return make_node$\big(m_{\text{var}}, \text{and_ite}(n, m_{\text{true}}), \text{and_ite}(n, m_{\text{false}})\big)$
  else
    return make_node$\big(n_{\text{var}}, \text{and_ite}(n_{\text{true}}, m_{\text{true}}), \text{and_ite}(n_{\text{false}}, m_{\text{false}})\big)$

Algorithm 1: The and_ite function

Require: two ROBDD nodes n and m
  function or_ite(n, m)
  if $n = \mathbf{0}$ **then**
    return m
  else if $m = \mathbf{0}$ **then**
    return n
  else if $n = \mathbf{1}$ **or** $m = \mathbf{1}$ **then**
    return 1
  else if $n_{\text{var}} \prec m_{\text{var}}$ **then**
    return make_node$\big(n_{\text{var}}, \text{or_ite}(n_{\text{true}}, m), \text{or_ite}(n_{\text{false}}, m)\big)$
  else if $m_{\text{var}} \prec n_{\text{var}}$ **then**
    return make_node$\big(m_{\text{var}}, \text{or_ite}(n, m_{\text{true}}), \text{or_ite}(n, m_{\text{false}})\big)$
  else
    return make_node$\big(n_{\text{var}}, \text{or_ite}(n_{\text{true}}, m_{\text{true}}), \text{or_ite}(n_{\text{false}}, m_{\text{false}})\big)$

Algorithm 2: The or_ite function

We have four terminal cases (note that there could be more or less: this choice is the result of statistical analysis):

If $i = \mathbf{1}$ the result is t.

If $i = \mathbf{0}$ the result is e.

If $t = \mathbf{1}$ and $e = \mathbf{0}$ the result is i.

If $t = e$ the result is t.

The implementation of operators with more than two arguments can be easily extended from the implementation of a generic binary operator, the main differences being that the splitting is performed on three nodes on each recursive call and the hash table used to avoid duplicate recursive calls uses a triple of ROBDD pointers as its key instead of a couple. Pseudo-code for this algorithm can be found in Algorithm 3.

3.2.6 Specializations of if/then/else

CORAL also provides some specializations of the `ite` operator, the most important being `var_ite` and `gen_var_ite`; their purpose is to provide a more efficient implementation of `ite` for some particular cases, which will occur in particular during the algorithm for the elimination of entailed variables that will be presented later.

For example, `gen_var_ite(i, t, e)` implements the particular case when i is a ROBDD whose root has **1** and **0** as its direct true and false successors respectively. `gen_var_ite_g` and `gen_var_ite_h` are further specializations of `gen_var_ite`: the first is used when we know that the root of i is labeled with a variable that is smaller than the one which labels the root of t, while the second is used when we know that the root of i is labeled with a variable that is smaller than the one which labels the root of e.

`var_ite(i, t, e)` and its further specializations `var_ite_g` and `var_ite_h` are analogous, but we add the hypothesis that the variable which labels the root of i does not occour in t and e.

Here's the list of all cases considered in the implementation of `gen_var_ite(f, g, h)`. Further informations, along with a correctness proof, can be found in [BS98]. For the sole purpose of writing these cases more simply, we consider $\mathbf{0}_{\text{var}}$ and $\mathbf{1}_{\text{var}}$ to be special variables that are greater than all other variables.

Case 1 If $f_{\text{var}} = g_{\text{var}} = h_{\text{var}}$,
return make_node$\big(f_{\text{var}}, g_{\text{true}}, h_{\text{true}}\big)$.

Case 2 If $f_{\text{var}} < g_{\text{var}} < h_{\text{var}}$ or $f_{\text{var}} < g_{\text{var}} = h_{\text{var}}$ or $f_{\text{var}} < h_{\text{var}} < g_{\text{var}}$,
return make_node$\big(f_{\text{var}}, g, h\big)$.

Case 3 If $f_{\text{var}} = g_{\text{var}} < h_{\text{var}}$,
return make_node$\big(f_{\text{var}}, g_{\text{true}}, h\big)$.

Require: three ROBDD nodes i, t and e
  function ite(i, t, e)
  if $i = \mathbf{1}$ **then**
    return t
  else if $i = \mathbf{0}$ **then**
    return e
  else if $t = \mathbf{1}$ **and** $e = \mathbf{0}$ **then**
    return i
  else if $t = e$ **then**
    return t
  else
    $x_{top} = \min\big(\{i_{\text{var}}, t_{\text{var}}, e_{\text{var}}\}\big)$
    if $x_{top} = i_{\text{var}}$ **then**
      $i_{newt} = i_{\text{true}}$
      $i_{newf} = i_{\text{false}}$
    else
      $i_{newt} = i_{newf} = i$
    if $x_{top} = t_{\text{var}}$ **then**
      $t_{newt} = t_{\text{true}}$
      $t_{newf} = t_{\text{false}}$
    else
      $t_{newt} = t_{newf} = t$
    if $x_{top} = e_{\text{var}}$ **then**
      $e_{newt} = e_{\text{true}}$
      $e_{newf} = e_{\text{false}}$
    else
      $e_{newt} = e_{newf} = e$
    return make_node$\big(x_{top}, \text{ite}(i_{newt}, t_{newt}, e_{newt}), \text{ite}(i_{newf}, t_{newf}, e_{newf})\big)$

Algorithm 3: The ite function

Case 4 If $f_{\text{var}} = h_{\text{var}} < g_{\text{var}}$,
return make_node$\big(f_{\text{var}}, g, h_{\text{false}}\big)$.

Case 5 If $g_{\text{var}} < f_{\text{var}} < h_{\text{var}}$,
return make_node$\big(g_{\text{var}}, gen_var_ite_h(f, g_{\text{true}}, h), gen_var_ite_h(f, g_{\text{false}}, h)\big)$.

Case 6 If $g_{\text{var}} < f_{\text{var}} = h_{\text{var}}$ or $g_{\text{var}} < h_{\text{var}} < f_{\text{var}}$,
return make_node$\big(g_{\text{var}}, gen_var_ite(f, g_{\text{true}}, h), gen_var_ite(f, g_{\text{false}}, h)\big)$.

Case 7 If $g_{\text{var}} = h_{\text{var}} < f_{\text{var}}$,
return make_node$\big(g_{\text{var}}, gen_var_ite(f, g_{\text{true}}, h_{\text{true}}), gen_var_ite(f, g_{\text{false}}, h_{\text{false}})\big)$.

Case 8 If $h_{\text{var}} < f_{\text{var}} < g_{\text{var}}$,
return make_node$\big(h_{\text{var}}, gen_var_ite_g(f, g, h_{\text{true}}), gen_var_ite_g(f, g, h_{\text{false}})\big)$.

Case 9 If $h_{\text{var}} < f_{\text{var}} = g_{\text{var}}$ or $h_{\text{var}} < g_{\text{var}} < f_{\text{var}}$,
return make_node$\big(h_{\text{var}}, gen_var_ite(f, g, h_{\text{true}}), gen_var_ite(f, g, h_{\text{false}})\big)$.

The set of cases for `var_ite` is a strict subset of these cases, obtained by removing the ones where $f_{\text{var}} = g_{\text{var}}$ or $f_{\text{var}} = h_{\text{var}}$ and replacing all calls to `gen_var_ite`, `gen_var_ite_h` and `gen_var_ite_g` with calls to `var_ite`, `var_ite_h` and `var_ite_g` respectively.

3.2.7 Variables on which a Boolean function depends

The algorithm `vars` which takes a ROBDD representing the Boolean function f and returns vars(f) can be easily implemented by visiting the ROBDD and collecting the variables that label the nodes. This algorithm is one of the few ROBDD algorithms in CORAL for which no caching is provided. Pseudo-code is given in Algorithm 4.

Require: a ROBDD node n
  function vars(n)
  if $n = \mathbf{0}$ **or** $n = \mathbf{1}$ **then**
    return $\varnothing$
  else
    return $\{n_{\text{var}}\} \cup \text{vars}(n_{\text{true}}) \cup \text{vars}(n_{\text{false}})$

Algorithm 4: The vars function

3.2.8 Entailed and disentailed variables

The algorithms `entailed_vars` and `disentailed_vars` (and their counterparts `entailed_vars_no_cache` and `disentailed_vars_no_cache`, which don't use caching) are used to compute true(f) and false(f). In order to compute the set of all entailed variables in a ROBDD, we start from its root node and follow these steps (the algorithm that looks for disentailed variables is very similar):

Step 1 If the false successor is **0** add the variable by which this node is labeled to the set and proceed to the true successor.

Step 2 If the true successor is **0** proceed to the false successor.

Step 3 If both successors are non-terminal nodes, start a new instance of the algorithm on both successors in order to obtain two new sets of variables: the resulting set is the union of the set that has already been computed by this instance with the intersection between these two new sets.

Step 4 Otherwise, stop.

The proof that this algorithm finds all and only the entailed variables can be found in [BS98].

Notice that we dont't consider the case when the ROBDD is **0**, as all variables in Vars are entailed and disentailed for $\bot$ so we would have to return an infinite set. Pseudo-code for both algorithms is given in Algorithm 5 and Algorithm 6.

Require: a ROBDD node n
  function entailed_vars(n)
  if $n_{\text{false}} = \mathbf{0}$ **then**
    return $\{n_{\text{var}}\} \cup$ entailed_vars(n_{true})
  else if $n_{\text{true}} = \mathbf{0}$ **then**
    return entailed_vars(n_{false})
  else if is_not_leaf(n_{true}) **and** is_not_leaf(n_{false}) **then**
    return entailed_vars(n_{true}) $\cap$ entailed_vars(n_{false})
  else
    return $\varnothing$

Algorithm 5: The entailed_vars function

Require: a ROBDD node n
  function disentailed_vars(n)
  if $n_{\text{true}} = \mathbf{0}$ **then**
    return $\{n_{\text{var}}\} \cup$ disentailed_vars(n_{false})
  else if $n_{\text{false}} = \mathbf{0}$ **then**
    return disentailed_vars(n_{true})
  else if is_not_leaf(n_{true}) **and** is_not_leaf(n_{false}) **then**
    return disentailed_vars(n_{true}) $\cap$ disentailed_vars(n_{false})
  else
    return $\varnothing$

Algorithm 6: The disentailed_vars function

3.2.9 Combined search for entailed, disentailed and equivalent variables

The algorithm `equivalent_entailed_disetailed_vars_no_cache` is used to find all equivalent, entailed and disentailed vars in a ROBDD by performing a single visit on it. The logic used for detecting entailed and disentailed variables is the same as the one of the previous algorithm, while for equivalent variables we say that the equivalent variables of a ROBDD having its root labeled with variable x are the ones that are equivalent for both its true and false successors, plus the equivalences given by the fact that x is equivalent to the variables that are both entailed in the true successor and disentailed in the false successor. This statement has been proven in [BS98]. We must also consider the fact that all entailed and disentailed variables are equivalent to each other: you can see how we handle this in Algorithm 7.

Note that when whenever we compare the current variable with the maximum variable in a set, we are exploiting the ordering property of ROBDDs.

3.2.10 Build a ROBDD representing some variable equivalences

Given an equivalence relation L over Vars, the algorithm `make_equivalent_vars` can be used to build the ROBDD representing the Boolean function that is true if and only if $\forall(x, y) \in L : x \Leftrightarrow y$. We define the current variable x_c and the set of true leaders T (it contains the leaders for L such that the branch of the ROBDD that we are currently building is reached only when all of these leaders are true); the first is initialized to the smallest variable in $\mathrm{dom}(L)$, while the second is initialized to $\varnothing$. The algorithm proceeds as follows:

Step 1 If x_c is undefined (which means that we have already processed all variables in $\mathrm{dom}(L)$) return **1**.

Step 2 Let x_n be the smallest variable in $\mathrm{dom}(L)$ that is greater than x_c and $x_l = \lambda_L(x_c)$. If $x_l = x_c$ (x_c is a leader for L) return the ROBDD having its root labeled with x_c, its false successor being the result of the recursive application of this algorithm with $x_c = x_n$ and its true successor being the result of the recursive application of this algorithm with x_n as the current variable and $T = T \cup \{x_c\}$.

Step 3 Otherwise, if $x_l \in T$, return the ROBDD having its root labeled with x_c, its true successor being the result of the recursive application of this algorithm with $x_c = x_n$ and its false successor being **0**.

Step 4 Otherwise, return the ROBDD having its root labeled with x_c, its false successor being the result of the recursive application of this algorithm with $x_c = x_n$ and its true successor being **0**.

3.2.11 Projection onto a set

The algorithm `project_set` is used to compute $f|_X$ given a ROBDD representing f and a set of variables X. We start from the root and follow these steps:

Require: a ROBDD node n
  function equivalent_entailed_disentailed_vars(n)
  $E = D = L = \varnothing$
  add_equiv_ent_disent_vars(n, E, D, L)

  function add_equiv_ent_disent_vars(n, E, D, L)
  if is_not_leaf(n) **then**
    if $n_{\text{true}} = \mathbf{0}$ **then**
      if $n_{\text{false}} \neq \mathbf{1}$ **then**
        add_equiv_ent_disent_vars($n_{\text{false}}, E, D, L$)
        $D = D \cup \{n_{\text{var}}\}$
        if $n_{\text{var}} \neq \max(D)$ **then**
          $L = L \wedge (n_{\text{var}}, \max(D))$
      else
        $D = D \cup \{n_{\text{var}}\}$
    else if $n_{\text{false}} = \mathbf{0}$ **then**
      if $n_{\text{true}} \neq \mathbf{1}$ **then**
        add_equiv_ent_disent_vars(n_{true}, E, D, L)
        $E = E \cup \{n_{\text{var}}\}$
        if $n_{\text{var}} \neq \max(E)$ **then**
          $L = L \wedge (n_{\text{var}}, \max(E))$
      else
        $E = E \cup \{n_{\text{var}}\}$
    else if $n_{\text{true}} \neq \mathbf{1}$ **and** $n_{\text{false}} \neq \mathbf{1}$ **then**
      $E_{\text{true}} = D_{\text{false}} = L_{\text{false}} = \varnothing$
      add_equiv_ent_disent_vars($n_{\text{true}}, E_{\text{true}}, D, L$)
      add_equiv_ent_disent_vars($n_{\text{false}}, E, D_{\text{false}}, L_{\text{false}}$)
      $E = E \cap E_{\text{true}}$
      $D = D \cap D_{\text{false}}$
      $L = L \vee L_{\text{false}}$
      $N = E_{\text{true}} \cap D_{\text{false}}$
      if $N \neq \varnothing$ **then**
        $L = L \wedge (n_{\text{var}}, \max(N))$

Algorithm 7: The equivalent_entailed_disentailed_vars function

Step 1 If the node is **1** or **0** return this node.

Step 2 If the set is empty or the variable which labels this node is greater than the maximum variable in the set return **1** (the reason for this is that all descendants of this node are necessarily labeled with a greater variable so they must be all projected away).

Step 3 If the variable by which this node is labeled is present in X call the algorithm recursively on the true and false successors: the result will be the ROBDD whose root is labeled by this variable and has the results of the two recursive calls as its true and false successor respectively.

Step 4 Otherwise this node must be projected away. We call the algorithm recursively on the true and false successors: the result will be the disjunction of the results of the two recursive calls, according to Equation 1.1.

The algorithm `project_threshold`, which projects away all variables in a ROBDD that are greater than a given variable, is analogous. The pseudo-code for both algorithms is given in Algorithm 8 and Algorithm 9.

Require: a ROBDD node n and a set of variables V
  function project_set(n, V)
  if is_leaf(n) **then**
    return n
  else if $\max(V) \prec n_{\text{var}}$ **or** $V = \varnothing$ **then**
    return 1
  else if $n_{\text{var}} \in V$ **then**
    return make_node$\big(n_{\text{var}}, \text{project_set}(n_{\text{true}}, V), \text{project_set}(n_{\text{false}}, V)\big)$
  else
    return or_ite$\big(\text{project_set}(n_{\text{true}}, V), \text{project_set}(n_{\text{false}}, V)\big)$

Algorithm 8: The project_set function

Require: a ROBDD node n and a threshold variable t
  function project_threshold(n, t)
  if is_leaf(n) **then**
    return n
  else if $t \prec n_{\text{var}}$ **then**
    return 1
  else
    return make_node$\big(n_{\text{var}}, \text{project_threshold}(n_{\text{true}}, t), \text{project_threshold}(n_{\text{false}}, t)\big)$

Algorithm 9: The project_threshold function

3.2.12 Shift renaming

The algorithm `shift_rename` takes an unsigned integer n and renames each variable v_i in a ROBDD to v_{i+n}; it is implemented easily by renaming the root then calling recursively on its true and false successors. Pseudo-code is given in Algorithm 10.

Require: a ROBDD node n and an offset s
  function shift_rename(n, s)
  if is_leaf(n) **then**
    return n
  else
    return make_node$\big(x_{\text{variable_index}(n)+s}$, shift_rename$(n_{\text{true}}, s)$,
      shift_rename$(n_{\text{false}}, s)\big)$

Algorithm 10: The shift_rename function

3.2.13 Renaming with respect to an equivalence relation

Suppose that we have a ROBDD n representing the Boolean function f and an equivalence relation R and we want to rename each variable in n to its leader in R. This operation is realized by the `rename` algorithm.

We start by defining the current variable x_c and the set of true leaders T. The first is initialized to x_0, while the second is initialized to $\varnothing$. The algorithm starts from the root of the ROBDD and proceeds as follows:

Step 1 If this node is **0** or **1** or it is labeled with a variable that is greater than the greatest variable in $\text{dom}(R)$, return this node.

Step 2 Let x_i be the variable by which this node is labeled. If there exists the minimum leader x_l for R such that $x_c \preceq x_l \prec x_i$ return the node that is labeled with x_l, has the result of the recursive application of this algorithm on this same node with x_{l+1} as the current variable and with $T = T \cup \{x_l\}$ as its true successor and has the result of the recursive application of this algorithm on this same node with x_{l+1} as the current variable and with the same old value of T as its false successor.

Step 3 If $x_i \notin \text{dom}(R)$ return the node that is labeled with x_i and has the results of the recursive applications of this algorithm on the true and false successors of this node with x_{i+1} as the current variable and with the same value of T as its true and false successors respectively.

Step 4 If $\lambda_R(x_i) = x_i$ return the node that is labeled with x_i, has the result of the recursive application of this algorithm on the true successor of this node with x_{i+1} as the current variable and with $T = T \cup \{x_i\}$ as its true successor and has the result of the recursive application of this algorithm on the false successor of this node with x_{i+1} as the current variable and with the same old value of T as its false successor.

Step 5 If $\lambda_R(x_i) \in T$ return the result of the recursive application of this algorithm on the true successor of this node with x_{i+1} as the current variable and with the same value of T.

Step 6 Here we have that $\lambda_R(x_i) \notin T$, so we return the result of the recursive application of this algorithm on the false successor of this node with x_{i+1} as the current variable and with the same value of T.

A correctness proof can be found, as usual, in [BS98], while the pseudo-code can be found in Algorithm 11.

Require: a ROBDD node n and an equivalence relation R
  function rename(n, R)
  $T = \varnothing$
  rename_impl$(n, R, 1, T)$

  function rename_impl(n, R, c, T)
  if is_leaf(n) **or** $\max(\mathrm{dom}(R)) \prec n_{\mathrm{var}}$ **then**
    return n
  else if $\exists \min\big(\{l \in \mathrm{leaders}(R) \,.\, x_c \preceq l \prec n_{\mathrm{var}}\}\big)$ **then**
    return make_node$(n_{\mathrm{var}}$, rename_impl$(n, R,$ variable_index$(n_{\mathrm{var}}) + 1, T \cup \{n_{\mathrm{var}}\})$,
                rename_impl$(n, R,$ variable_index$(n_{\mathrm{var}}) + 1, T))$
  else if $n_{\mathrm{var}} \notin \mathrm{dom}(R)$ **then**
    return make_node$(n_{\mathrm{var}}$, rename_impl$(n_{\mathrm{true}}, R,$ variable_index$(n_{\mathrm{var}}) + 1, T)$,
                rename_impl$(n_{\mathrm{false}}, R,$ variable_index$(n_{\mathrm{var}}) + 1, T))$
  else if $\lambda_R(n_{\mathrm{var}}) = n_{\mathrm{var}}$ **then**
    return make_node$(n_{\mathrm{var}}$,
                rename_impl$(n_{\mathrm{true}}, R,$ variable_index$(n_{\mathrm{var}}) + 1, T \cup \{n_{\mathrm{var}}\})$,
                rename_impl$(n_{\mathrm{false}}, R,$ variable_index$(n_{\mathrm{var}}) + 1, T))$
  else if $\lambda_R(n_{\mathrm{var}}) \in T$ **then**
    return rename_impl$(n_{\mathrm{true}}, R,$ variable_index$(n_{\mathrm{var}}) + 1, T)$
  else
    return rename_impl$(n_{\mathrm{false}}, R,$ variable_index$(n_{\mathrm{var}}) + 1, T)$

Algorithm 11: The rename function

3.2.14 Elimination of entailed and disentailed variables

Let n be the ROBDD representing the Boolean function f and S a finite subset of Vars. The algorithms `elim_true_vars` and `elim_false_vars` can be used to compute $f[1/S]$ and $f[0/S]$ respectively. In particular, if S is the set of the entailed variables for f, then the result of the application of `elim_true_vars` will be the ROBDD obtained by removing all entailed variables from n.

The algorithm `elim_true_vars` (`elim_false_vars` is analogous) starts from the root of the ROBDD and follows these steps:

Step 1 If this node if **0** or **1** or it is labeled with a variable that is greater than all the variables in S return this node.

Step 2 Let x be the variable by which this node is labeled. If $x \in S$ return the result of the recursive application of this algorithm on the true successor.

Step 3 Here we have $x \notin S$, so we simply return the node that is labeled with x and has the results of the recursive applications of this algorithm on the true and false successors as its true and false successors respectively.

Pseudo-code for the two algorithms is given in Algorithm 12 and Algorithm 13.

Require: a ROBDD node n and a set of variables E
  function elim_true_vars(n, E)
  if is_leaf(n) **or** $\max(E) \prec n_{\text{var}}$ **then**
    return n
  else if $n_{\text{var}} \in E$ **then**
    return elim_true_vars(n_{true}, E)
  else
    return make_node$\big(n_{\text{var}}$, elim_true_vars(n_{true}, E), elim_true_vars$(n_{\text{false}}, E)\big)$

Algorithm 12: The elim_true_vars function

Require: a ROBDD node n and a set of variables D
  function elim_false_vars(n, D)
  if is_leaf(n) **or** $\max(D) \prec n_{\text{var}}$ **then**
    return n
  else if $n_{\text{var}} \in D$ **then**
    return elim_false_vars(n_{false}, D)
  else
    return make_node$\big(n_{\text{var}}$, elim_false_vars(n_{true}, D), elim_false_vars$(n_{\text{false}}, D)\big)$

Algorithm 13: The elim_false_vars function

3.2.15 Elimination of equivalent variables

Let n be the ROBDD representing the Boolean function f and L the (finite) equivalence relation over Vars which represents equiv(f) (and can be obtained by applying the algorithm presented in 3.2.9 on n). The algorithm `squeeze_equivalent_vars` can be used to remove all nodes that are labeled with a variable that is not a leader for L from n.

The implementation is quite simple: we visit the ROBDD as usual. When we encounter a node that is labeled with a leader for L the result is simply a node having the same label and the results of the recursive calls on the original node's successors as its successors, otherwise one of the successors of the node must be **0** and we return the result of the recursive call of the algorithm on the other node (thus projecting away this node). You can find the pseudo-code in Algorithm 14.

Require: a ROBDD node n and an equivalence relation L
  function squeeze_equivalent_vars(n, L)
  if is_leaf(n) **or** max(dom(L)) $\prec n_{\text{var}}$ **then**
    return n
  else
    if $n_{\text{var}} \notin \text{dom}(L)$ **or** $\lambda_L(n_{\text{var}}) = n_{\text{var}}$ **then**
      return make_node$\big(n_{\text{var}}$, squeeze_equivalent_vars(n_{true}, L),
                squeeze_equivalent_vars(n_{false}, L)$\big)$
    else
      if $n_{\text{true}} = \mathbf{0}$ **then**
        return squeeze_equivalent_vars(n_{false}, L)
      else
        return squeeze_equivalent_vars(n_{true}, L)

Algorithm 14: The squeeze_equivalent_vars function

3.2.16 Combined search and elimination of entailed and disentailed variables

Suppose that we have a ROBDD n for Boolean function f and we know some (but not necessarily all) entailed variables for f, i.e. we know $E \subset \text{true}(f)$. The algorithm `elim_search_true_vars` can be used to eliminate all entailed variables in E from n while adding some other unknown entailed variables for f to E. Repeated application of this algorithm on the same ROBDD is guaranteed to reach a fixpoint where E contains all and only the entailed variables for f and n does not have any entailed variable.

Algorithm `elim_search_false_vars` is the analogous for disentailed variables. Pseudo-codes for both can be found in Algorithm 15 and Algorithm 16 (notice that they return a pair (n, S)).

3.2.17 Forcing a set of variables to true or false

Given a ROBDD n representing the Boolean function f and a finite set of variables $S = x_1, x_2, \ldots, x_n$, the algorithm `and_vs_true` computes the ROBDD representing the Boolean function $f \wedge x_1 \wedge x_2 \wedge \ldots \wedge x_n$ (the `algorithm and_vs_false` which computes $f \wedge \overline{x_1} \wedge \overline{x_2} \wedge \ldots \wedge \overline{x_n}$ is very similar to this one). We traverse the ROBDD starting from the root node , initialize the current variable $x_c = x_1$, and apply the following steps:

Require: an ROBDD node n and a set of variables E
  function elim_search_true_vars(n, E)
  if is_leaf(n) **then**
    return (n, E)
  else if $n_{\text{var}} \in E$ **or** $n_{\text{false}} = \mathbf{0}$ **then**
    $E = E \cup \{n_{\text{var}}\}$
    return (elim_search_true_vars$(n_{\text{true}}, E), E)$
  else if $E = \varnothing$ **or** $\max(E) \prec n_{\text{var}}$ **then**
    return (n, E)
  else
    return (make_node$\big(n_{\text{var}}$, elim_true_vars(n_{true}, E), elim_true_vars$(n_{\text{false}}, E)\big), E)$

Algorithm 15: The elim_search_true_vars function.

Require: an ROBDD node n and a set of variables D
  function elim_search_false_vars(n, D)
  if is_leaf(n) **then**
    return (n, D)
  else if $n_{\text{var}} \in D$ **or** $n_{\text{true}} = \mathbf{0}$ **then**
    $D = D \cup \{n_{\text{var}}\}$
    return (elim_search_false_vars$(n_{\text{false}}, D), D)$
  else if $D = \varnothing$ **or** $\max(D) \prec n_{\text{var}}$ **then**
    return (n, D)
  else
    return (make_node$\big(n_{\text{var}}$, elim_false_vars(n_{true}, D), elim_false_vars$(n_{\text{false}}, D)\big), D)$

Algorithm 16: The elim_search_false_vars function.

Step 1 If the node is **0** return this node.

Step 2 If the node is **1** build and return the ROBDD for Boolean function $x_c \wedge x_{c+1} \wedge \ldots \wedge x_n$ (how to build it is trivial and is shown in Algorithm 17).

Step 3 If the variable x by which this node is labeled is smaller than x_c return the ROBDD whose root is labeled by x and whose true and false successors are the results of the application of this algorithm on the true and false successors of this node respectively.

Step 4 Let t be this node if $x_c \preceq x$ or its true successor otherwise. If x_c is the greatest variable in G, then return the ROBDD having its root labeled with c, t as its true successor and **0** as its false successor; otherwise return the ROBDD having its root labeled with c, the result of the recursive application of this algorithm on t as its true successor and **0** as its false successor.

You can find the pseudo-code for both algorithms in Algorithm 17 and Algorithm 18.

Require: a ROBDD node n and a set of variables T
  function and_vs_true(n, T)
  avt_impl($n, T, 1$)

  function avt_impl(n, T, c)
  if $n = \mathbf{0}$ **then**
    return 0
  else if $n = \mathbf{1}$ **then**
    $n_{res} = \mathbf{1}$
    while $x_c \preceq \max(T)$ **do**
      $n_{res} = \text{make\_node}\big(\max(T), n_{res}, \mathbf{0}\big)$
      $T = T \setminus \big\{\max(T)\big\}$
    return n_{res}
  else if $n_{\text{var}} \prec x_c$ **then**
    return $\text{make\_node}\big(n_{\text{var}}, \text{avt\_impl}(n_{\text{true}}, T, c), \text{avt\_impl}(n_{\text{false}}, T, c)\big)$
  else
    if $x_c \preceq n_{\text{var}}$ **then**
      $n_{next} = n$
    else
      $n_{next} = n_{\text{true}}$
    if $\exists x_{next} = \min\big(\{x \in T \,.\, x_c \prec x\}\big)$ **then**
      return $\text{make\_node}(x_c, \text{avt\_impl}(n_{next}, T, \text{variable\_index}(x_{next})), \mathbf{0})$
    else
      return $\text{make\_node}(x_c, n_{next}, \mathbf{0})$

Algorithm 17: The and_vs_true function

Require: a ROBDD node n and a set of variables F
  function and_vs_false(n, F)
  avf_impl($n, F, 1$)

  function avf_impl(n, F, c)
  if $n = \mathbf{0}$ **then**
    return n
  else if $n = \mathbf{1}$ **then**
    $n_{res} = \mathbf{1}$
    while $x_c \preceq \max(F)$ **do**
      $n_{res} = \text{make\_node}\big(\max(F), \mathbf{0}, n_{res}\big)$
      $F = F \setminus \big\{\max(F)\big\}$
    return n_{res}
  else if $n_{\text{var}} \prec x_c$ **then**
    return $\text{make\_node}\big(n_{\text{var}}, \text{avf\_impl}(n_{\text{true}}, F, c), \text{avf\_impl}(n_{\text{false}}, F, c)\big)$
  else
    if $x_c \preceq n_{\text{var}}$ **then**
      $n_{next} = n$
    else
      $n_{next} = n_{\text{false}}$
    if $\exists x_{next} = \min\big(\{x \in F \,.\, x_c \prec x\}\big)$ **then**
      return $\text{make\_node}(x_c, \mathbf{0}, \text{avf\_impl}(n_{next}, F, \text{variable\_index}(x_{next})))$
    else
      return $\text{make\_node}(x_c, \mathbf{0}, n_{next})$

Algorithm 18: The and_vs_false function

3.2.18 Forcing some variables to be equivalent in a positive function

The algorithm `expand_equivalent_vars` can be used to perform efficiently the inverse operation for the one presented in 3.2.15, that is conjuncting a Boolean function with the equivalence conditions given by an equivalence relation, for the case when $f \in Pos$.

Given a ROBDD representing f and the equivalence relation L, we define the current variable x_c and the set of true leaders T; the first is initialized to the smallest variable in $\mathrm{dom}(L)$, while the second is initialized to $\varnothing$. Afterwards, we start from the root of the ROBDD and follow these steps:

Step 1 If this node is **1** return the result of the application of the algorithm `make_equivalent_vars` (presented in 3.2.10) on the same relation L, using the same values for x_c and T.

Step 2 If this node if **0** return **0**.

Step 3 If x_c is undefined (which means that we have already processed all variables in $\mathrm{dom}(L)$) return the current node.

Step 4 Let x be the variable by which this node is labeled. If $x \prec x_c$ return the node labeled with x and having the results of the recursive calls of this algorithm (with the same values for x_c and T) on the true and false successors of the current node as its true and false successors respectively.

Step 5 Let x_n be the smallest variable in $\mathrm{dom}(L)$ that is greater than x_c. If $x = x_c$ then x_c must be a leader for L, so we return the node labeled with x_c, having the result of the recursive application of this algorithm on the true successor of this node with x_n as the current variable and with $T = T \cup \{x_c\}$ as its true successor and the result of the recursive application of this algorithm on the false successor of this node with x_n as the current variable and with the same old value of T as its false successor.

Step 6 Otherwise, let $x_l = \lambda_L(x_c)$. If $x_l = x_c$ then x_c is a leader for L so we return the node labeled with x_c, having the result of the recursive application of this algorithm on this same node with x_n as the current variable and with $T = T \cup \{x_c\}$ as its true successor and the result of the recursive application of this algorithm on this same node with x_n as the current variable and with the same old value of T as its false successor.

Step 7 If $x_l \in T$ return the node labeled with x_c, having the result of the recursive application of this algorithm on this same node with x_n as the current variable and with the same value of T as its true successor and **0** as its false successor.

Step 8 Otherwise, return the node labeled with x_c, having the result of the recursive application of this algorithm on this same node with x_n as the current variable and with the same value of T as its false successor and **0** as its true successor.

3.2.19 Exotic operators

CORAL also provides a few more specific ROBDD algorithms for the purpose of building (efficiently) some ROBDDs representing particular classes of Boolean functions that are often encountered in groundness analysis.

Functions of type $v \Longrightarrow v_1 \wedge v_2 \wedge \ldots \wedge v_n$

ROBDDs for this class of functions are built by the `if_conj` algorithm, which takes v and the set of variables $V = \{v_1, v_2, \ldots v_n\}, v \notin V$ as its arguments. If all variables in V are greater than v in the ordering the task is trivial, as the result is simply a ROBDD whose root is labeled by v, has $\mathbf{1}$ as its false successor and has the ROBDD representing $v_1 \wedge v_2 \wedge \ldots \wedge v_n$ as its true successor: that's why we start by considering only the grater variables and building the ROBDD for them. The smaller variables are handled later, as you can see in Algorithm 19.

Require: a variable v and a set of variables V
  function if_conj(v, V)
  $n_v = \text{and\_vs\_true}\big(\mathbf{1}, \{x \in V \,.\, v \prec x\}\big)$
  $n_v = \text{make\_node}\big(v, n_v, \mathbf{1}\big)$
  $S = \{x \in V \,.\, x \prec v\}$
  if $S = \varnothing$ **then**
    return n_v
  else
    $f_v = \text{make\_node}\big(v, \mathbf{0}, \mathbf{1}\big)$
    $curr = prev = n_v$
    while $S \neq \varnothing$ **do**
      $m = \max(S)$
      $curr = \text{make\_node}\big(m, prev, f_v\big)$
      $prev = curr$
      $S = S \setminus \{m\}$
    return *curr*

Algorithm 19: The if_conj function.

Functions of type $v \Longleftrightarrow v_1 \wedge v_2 \wedge \ldots \wedge v_n$

ROBDDs for this class of functions are built by the `iff_conj` algorithm, which takes v and the set of variables $V = \{v_1, v_2, \ldots v_n\}, v \notin V$ as its arguments. As for `if_conj`, the construction is trivial if all variables in V are greater than v in the ordering, as the result is simply a ROBDD whose root is labeled by v, has the ROBDD representing $\neg v_1 \vee \neg v_2 \vee \ldots \vee \neg n_n$ as its false successor and has the ROBDD representing $v_1 \wedge v_2 \wedge \ldots \wedge v_n$ as its true successor: that's why, as for `if_conj`, we start by considering only the greater variables and handle the smaller ones later. Pseudo-code is given in Algorithm 20.

Require: a variable v and a set of variables V
  function iff_conj(v, V)
  $G = \{x \in V \,.\, v \prec x\}$
  $t_v = \text{and\_vs\_true}(\mathbf{1}, G)$
  $f_{curr} = f_{prev} = \mathbf{0}$
  while $G \neq \varnothing$ **do**
    $m = \max(G)$
    $f_{curr} = \text{make\_node}(m, f_{prev}, \mathbf{1})$
    $f_{prev} = f_{curr}$
    $G = G \setminus \{m\}$
  $n_v = \text{make\_node}(v, t_v, f_{curr})$
  $S = \{x \in V \,.\, x \prec v\}$
  if $S = \varnothing$ **then**
    return n_v
  else
    $f_v = \text{make\_node}(v, \mathbf{0}, \mathbf{1})$
    $curr = prev = n_v$
    while $S \neq \varnothing$ **do**
      $m = \max(S)$
      $curr = \text{make\_node}(m, prev, f_v)$
      $prev = curr$
      $S = S \setminus \{m\}$
    return *curr*

Algorithm 20: The iff_conj function.

An even more exotic class of functions

The `conj` algorithm takes a set of variables $V = \{v_1, v_2, \ldots v_n\}$ as its argument and builds the ROBDD for the Boolean function that evaluates to $false$ if and only if one and only one variable in V is false. This function can be viewed as:

$$(v_0 \Longleftarrow v_1 \wedge v_2 \wedge \ldots \wedge v_n) \wedge (v_1 \Longleftarrow v_0 \wedge v_2 \wedge \ldots \wedge v_n) \wedge \ldots \wedge (v_n \Longleftarrow v_0 \wedge v_1 \wedge \ldots \wedge v_{n-1})$$

Since we must conjunct a set of clauses, we start by building the ROBDD for the first clause, then we build the ROBDD for the second one and conjunct it with the first one using `and_ite`, then we conjunct the result with the ROBDD for the third one and so on.

The only problem we still have to solve is how to build the ROBDD for a Boolean function of type $v \Longleftarrow t_1 \wedge t_2 \wedge \ldots \wedge t_n$. As with the two previous algorithms, the solution is trivial when the implied variable is greater than all the others: the result is simply a ROBDD whose root is labeled with v, has $\mathbf{1}$ as its true successor and has the ROBDD for $\neg t_1 \vee \neg t_2 \vee \ldots \vee \neg t_n$ as its false successor. So we start as usual by considering only the greater variables and building the corresponding ROBDD; afterwards we handle the smaller variables as shown in Algorithm 21.

Require: a set of variables V
  function conj(V)
  $n = \mathbf{1}$
  $T = V$
  while $T \neq \varnothing$ **do**
    $c = \min(T)$
    $f_{curr} = f_{prev} = \mathbf{0}$
    $G = \{x \in V \,.\, c \prec x\}$
    while $G \neq \varnothing$ **do**
      $m = \max(G)$
      $f_{curr} = \text{make\_node}(m, f_{prev}, \mathbf{1})$
      $f_{prev} = f_{curr}$
      $G = G \setminus \{m\}$
    $n_c = \text{make\_node}(c, \mathbf{1}, f_{curr})$
    $n_{curr} = n_{prev} = n_c$
    $S = \{x \in V \,.\, x \prec c\}$
    while $S \neq \varnothing$ **do**
      $m = \max(G)$
      $n_{curr} = \text{make\_node}(m, n_{prev}, \mathbf{1})$
      $n_{prev} = n_{curr}$
      $S = S \setminus \{m\}$
    $n = \text{and\_ite}(n, n_{curr})$
    $T = T \setminus \{c\}$
  return n

Algorithm 21: The conj function.

4 Policy-based composite representations

4.1 Composite representations of Boolean functions

4.1.1 Beyond ROBDDs

Statistical analysis in CHINA (see [BS98]) has shown that a very consistent part of ROBDDs was used to store informations about entailed and equivalent variables: this has led to the idea that we can keep ROBDDs much smaller (thus improving the efficiency of most algorithms) by keeping the information on entailed and equivalent variables in a separate `Bit_Set` and an `Equivalence_Relation` respectively. This can be extended to disentailed variables by using an additional `Bit_Set` if some other application needs it. Formally, we can represent the Boolean function $f \in \mathcal{F}$ with four elements $n \in \mathcal{D}$, $E, D \in \wp_{\mathrm{f}}(\mathrm{Vars})$ and $L \in \mathcal{L}$ such that:

$$\mathrm{vars}(\llbracket n \rrbracket_{\mathcal{D}}) \cap E = \mathrm{vars}(\llbracket n \rrbracket_{\mathcal{D}}) \cap D = E \cap D = \varnothing \tag{4.1}$$

(which means that the function represented by n must not depend on any variable in E or D and a variable cannot be both entailed and disentailed),

$$\mathrm{dom}(L) \cap \mathrm{vars}(\llbracket n \rrbracket_{\mathcal{D}}) \subseteq \mathrm{leaders}(L) \tag{4.2}$$

(which means that non-leaders for L must not occur in the ROBDD n),

$$\mathrm{dom}(L) \cap E = \mathrm{dom}(L) \cap D = \varnothing \tag{4.3}$$

(which means that all variables in L must not occur in E and D)

$$n = \mathbf{0} \Longrightarrow E = D = L = \varnothing \tag{4.4}$$

(which imposes a canonical representation for $\bot$)
and

$$f = \llbracket n \rrbracket_{\mathcal{D}} \wedge \bigwedge_{e \in E} e \wedge \bigwedge_{d \in D} \overline{d} \wedge \bigwedge_{(v_1, v_2) \in L} (v_1 \iff v_2) \tag{4.5}$$

If conditions 4.1, 4.2, 4.3 and 4.4 are all met, we say that the representation is *normalized*; otherwise we say that the representation is *not normalized*. For both cases the semantics is given by Equation 4.5. These equations can be trivially extended to the cases where we decide not to use all of the last three components. An algorithm for converting a non-normalized represetation to a normalized one will be given in 4.2.2.

4.1.2 Implementation policies

The `New_Boolean_Expression` class supports many different internal representations for Boolean functions:

The information on entailed variables may be stored in the ROBDD or kept separate.

The information on disentailed variables may be stored in the ROBDD or kept separate.

The information on equivalent variables may be stored in the ROBDD or kept separate.

Given the fact that some algorithms can be optimized when performed on positive Boolean functions, we also support the option of representing only positive functions so that the optimized algorithms can be used. Finally, we give the option to let CORAL keep the internal representation normalized automatically or to let the user do it when he desires. The class interface should be the same for all cases.

The trivial implementation would be to use 2^5 different classes. This of course is unacceptable for three main reasons: it would be extremely unelegant, the effort required to keep their interfaces coherent would be too big and it would be extremely hard to expand the library to provide new options in the future, as the number of classes would grow exponentially in the number of options. Our solution is to apply *policy-based class design* to `New_Boolean_Expression`: the implementation options are specified by small policy classes that contain only five static bool data members:

explicit_entailed_vars is true if we represent entailed variables explicitly.

explicit_disentailed_vars is true if we represent disentailed variables explicitly.

explicit_equivalent_vars is true if we represent equivalent variables explicitly.

lazy_normalization is true if we let the user normalize the internal representation manually.

positive_functions_only is true if we represent only positive functions and apply the optimized algorithms for them.

Listing 4.1 shows the implementation policy which represents only positive functions by keeping the information on entailed and equivalent variables separate (but not on disentailed variables) and utilizes automatic normalization.

`New_Boolean_Expression` is a templatic class that takes an implementation policy type `Policy` as its template argument and is also derived from its template argument type. This allows its methods to adopt different behaviors depending on the chosen policy by checking the values of its static members. This solution is shown in Listing 4.2 along with the trick that is used in order to have different data members according to the chosen internal representation, which consists of using arrays whose lengths can be 0 or 1 depending on the values of the static policy members. The listing also shows the private Boolean data member `normalized` that is used to record whether the internal representation has already been normalized or not so that we can avoid some unnecessary normalizations.

```
struct Record_Positive_Entailed_Equivalent {
  static const bool explicit_entailed_vars = true;
  static const bool explicit_disentailed_vars = false;
  static const bool explicit_equivalent_vars = true;
  static const bool lazy_normalization = false;
  static const bool positive_functions_only = true;
};
```

Listing 4.1: An example of implementation policy

```
template <class Policy>
class China_Original_ROBDD_Library::New_Boolean_Expression
      : public Policy {

public:

...

private:

  // The ROBDD
  ROBDD_Ptr residue;

  // The entailed variables
  Bit_Set entailed_vars[Policy::explicit_entailed_vars ? 1 : 0];

  // The disentailed variables
  Bit_Set disentailed_vars[Policy::explicit_disentailed_vars ? 1 : 0];

  // The equivalent variables
  Equivalence_Relation equivalent_vars[Policy::explicit_equivalent_vars ?
      1 : 0];

  // True when the internal representation is normalized
  bool normalized;
...

};
```

Listing 4.2: Templatic definition of class **New_Boolean_Expression**

4.2 Algorithms with composite representations

4.2.1 Normalization

All algorithms in this section will be described only for the case when entailed, disentailed and equivalent variables are all represented explicitly.

The first and most important algorithm is the *normalization algorithm*, which allows us to transform an internal representation where conditions 4.1, 4.2, 4.3 and 4.4 are not necessarily met to one where they are all met without changing the semantics given by Equation 4.5. Given this algorithm, all operations on composite representations (with an arbitrary number of arguments) can be implemented, once we have a corresponding implementation on ROBDDs, by this simple algorithm:

Denormalization *Denormalize* the representation, which means moving, for all the arguments of the operator, all informations on entailed, disentailed and equivalent variables that are stored in E, D and L into the ROBDD n.

Application Apply the corresponding operator on ROBDDs to the denormalized ROBDDs computed in the previous step.

Normalization Normalize the ROBDD computed in the previous step, which means moving all informations on entailed, disentailed and equivalent variables that are stored in the ROBDD into three separate components as described previously.

Of course this trivial implementation is unacceptable, as it goes against the principle of keeping the ROBDDs as small as possible when applying an operator on them in order to improve efficiency. That is why we will present better implementations for these operators right after describing our normalization algorithm.

The result of the application of the normalization function on the composite representation $\langle n, E, D, L\rangle$ will be denoted as $\eta(\langle n, E, D, L\rangle)$.

4.2.2 Implementing the normalization algorithm

CORAL's generic normalization algorithm is an adaptation of an old normalization algorithm that was a part of China and is described in [BS98]. It consists of these steps, which must be applied repeatedly until a fixpoint is reached:

Step 1 If $n = \mathbf{0}$ the final result of the algorithm is $\bot$.

Step 2 Look for all the equivalence classes in L that contain at least one variable in E and add all their variables to E. Do the same for D.

Step 3 Remove all the equivalence classes that were found in the previous step from L.

Step 4 Apply the `elim_true_vars` algorithm on n with E as its argument, thus eliminating all variables in E from n.

Step 5 Apply the `elim_false_vars` algorithm on n with D as its argument, thus eliminating all variables in D from n.

Step 6 Find all entailed variables in n using the algorithm `entailed_vars` and add them to E.

Step 7 Find all disentailed variables in n using the algorithm `disentailed_vars` and add them to D.

Step 8 If $E \cap D \neq \varnothing$, the final result of the algorithm is $\bot$.

Step 9 Find all equivalent variables in n by using the algorithm `equivalent_vars`, put this information in an equivalence relation T and do the assignment $L = L \wedge T$.

Step 10 Eliminate all informations on equivalent variables from n by applying the algorithm `squeeze_equivalent_vars` with T as its argument.

Step 11 Rename each variable in n to its leader in L by applying the `rename` algorithm.

This algorithm respects the loop invariant which states that after each iteration the semantics given in Equation 4.5 is preserved. Pseudo-code for the normalization algorithm can be found in Algorithm 22.

CORAL also implements a simpler, more efficient specialization of this algorithm for positive functions; in this case, of course, we must not consider disentailed variables. A pseudo-code for this specialized algorithm is shown in Algorithm 23.

4.2.3 Conjunction

The algorithms for conjunction and disjunction are an adaptation of the ones presented in [BS98]. Given two (not necessarily normalized) composite representations $\langle n_1, E_1, D_1, L_1\rangle$ and $\langle n_2, E_2, D_2, L_2\rangle$, we compute their conjunction as:

$$\langle n_1', E_1', D_1', L_1'\rangle = \eta(\langle n_1, E_1 \cup E_2, D_1 \cup D_2, L_1 \wedge L_2\rangle)$$

$$\langle n_2', E_2', D_2', L_2'\rangle = \eta(\langle n_2, E_1 \cup E_2, D_1 \cup D_2, L_1 \wedge L_2\rangle)$$

$$\langle n_1, E_1, D_1, L_1\rangle \wedge \langle n_2, E_2, D_2, L_2\rangle = \eta(\langle n_1' \wedge n_2', E_1' \cup E_2', D_1' \cup D_2', L_1' \wedge L_2'\rangle)$$

Notice that the initial collection of entailed, disentailed and equivalent variables allows for greater reduction of the ROBDDs when normalization is called (and recall that the $\wedge$ operator on equivalence relations represents the transitive closure of the union).

CORAL also provides a function `meet_all_assign` (and the corresponding `meet_all_up_assign` that utilizes upward approximation) that conjuncts more than two Boolean functions at the same time. In this case we start by collecting the entailed, disentailed and equivalent variables of ALL functions in order to be able to reduce the ROBDDs

Require: (n, E, D, L)
  function normalize(n, E, D, L)
  while $n_{old} \neq n$ **or** $E_{old} \neq E$ **or** $D_{old} \neq D$ **or** $L_{old} \neq L$ **do**
    if $n = \mathbf{0}$ **then**
      return $(\mathbf{0}, \varnothing, \varnothing, \varnothing)$
    $n_{old} = n$
    $E_{old} = E$
    $D_{old} = D$
    $L_{old} = L$
    $E = E \cup \left\{ y \in \text{Vars} \;\middle|\; \exists x \in E \,.\, (x, y) \in L \right\}$
    $D = D \cup \left\{ y \in \text{Vars} \;\middle|\; \exists x \in D \,.\, (x, y) \in L \right\}$
    $L = L \setminus \left\{ (x, y) \;\middle|\; x \in E \text{ or } x \in D \text{ or } y \in E \text{ or } y \in D \right\}$
    $n = \text{elim\_true\_vars}(n, E)$
    $n = \text{elim\_false\_vars}(n, D)$
    $E = E \cup \text{entailed\_vars}(n)$
    $D = D \cup \text{disentailed\_vars}(n)$
    if $E \cap D \neq \varnothing$ **then**
      return $(\mathbf{0}, \varnothing, \varnothing, \varnothing)$
    $T = \text{equivalent\_vars}(n)$
    $L = L \wedge T$
    $n = \text{squeeze\_equivalent\_vars}(n, T)$
    $n = \text{rename}(n, L)$
  return (n, E, D, L)

Algorithm 22: The normalize function

Require: (n, E, L)
  function normalize_positive(n, E, L)
  $n_{new} = n$
  $n_{old} = NULL$
  while $n_{old} \neq n_{new}$ **do**
    $(n_{old}, E, L) =$ i_normalize1(n_{new}, E, L)
    $n_{new} =$ rename(n_{old}, L)
  return (n_{new}, E, L)

  function i_normalize1(n, E, L)
  if $n =$ **1 then**
    $E = E \cup \{y \in \text{Vars} \mid \exists x \in E \,.\, (x, y) \in L\}$
    $L = L \setminus \{(x, y) \mid x \in E \text{ or } y \in E\}$
    return (n, E, L)
  else
    $n_{new} = n$
    $n_{old} = NULL$
    $L_{new} = L$
    $L_{old} = NULL$
    while $n_{old} \neq n_{new}$ **or** $L_{old} \neq L_{new}$ **do**
      $n_{old} = n_{new}$
      $L_{old} = L_{new}$
      $E = E \cup$ entailed_vars(n)
      $(n_{new}, E) =$ elim_search_true_vars(n_{new}, E)
      $E = E \cup \{y \in \text{Vars} \mid \exists x \in E \,.\, (x, y) \in L_{new}\}$
      $L_{new} = L_{new} \setminus \{(x, y) \mid x \in E \text{ or } y \in E\}$
    $T =$ equivalent_vars(n_{new})
    if $T \neq \varnothing$ **then**
      $n_{new} =$ squeeze_equivalent_vars(n_{new}, T)
      $L_{new} = L_{new} \wedge T$
    return (n_{new}, E, L_{new})

Algorithm 23: The normalize_positive function

more. Formally, given m composite representations $\langle n_1, E_1, D_1, L_1\rangle, \ldots, \langle n_m, E_m, D_m, L_m\rangle$, their conjunction can be computed, for each $i \in \mathbb{N} : 1 \leq i \leq m$, by:

$$\langle n'_i, E'_i, D'_i, L'_i\rangle = \eta\big(\langle n_i, \bigcup_{j=1}^{m} E_j, \bigcup_{j=1}^{m} D_j, \bigwedge_{j=1}^{m} L_j\rangle\big),$$

$$\langle n_1, E_1, D_1, L_1\rangle \wedge \ldots \wedge \langle n_m, E_m, D_m, L_m\rangle = \eta\big(\langle \bigwedge_{j=1}^{m} n'_j, \bigcup_{j=1}^{m} E'_j, \bigcup_{j=1}^{m} D'_j, \bigwedge_{j=1}^{m} L'_j\rangle\big).$$

The actual implementation, in fact, does not utilize this exact expression: whenever we perform normalization (except, of course, the final normalization) we actually avoid looking for all entailed, disentailed and equivalent variables in the ROBDDs; instead we just reduce the current ROBDD using the information on entailed, disentailed and equivalent variables that we have collected so far by using algorithms `elim_search_true_vars`, `elim_search_false_vars` and `rename`. If `elim_search_true_vars` or `elim_search_false_vars` find new entailed or disentailed variables then they are added to the sets of collected variables. We also conjunct each ROBDD immediately after this partial reduction instead of reducing the remaining ones before.

4.2.4 Disjunction

Given the same arguments as conjunction, disjunction can be performed as:

$$\langle n_1, E_1, D_1, L_1\rangle \vee \langle n_2, E_2, D_2, L_2\rangle = \langle n'_1 \vee n'_2, E_1 \cap E_2, D_1 \cap D_2, L'_1 \vee L'_2\rangle$$

where:

$$L'_1 = L_1 \wedge \bigwedge_{\substack{(x,y)\in E_1\setminus E_2 \\ x\prec y}} \big\{(x,y)\big\} \wedge \bigwedge_{\substack{(x,y)\in D_1\setminus D_2 \\ x\prec y}} \big\{(x,y)\big\},$$

$$L'_2 = L_2 \wedge \bigwedge_{\substack{(x,y)\in E_2\setminus E_1 \\ x\prec y}} \big\{(x,y)\big\} \wedge \bigwedge_{\substack{(x,y)\in D_2\setminus D_1 \\ x\prec y}} \big\{(x,y)\big\},$$

$$E'_1 = \big\{\, \lambda_{L'}(x) \;\big|\; x \in E_1 \setminus E_2 \,\big\}, E'_2 = \big\{\, \lambda_{L'}(x) \;\big|\; x \in E_2 \setminus E_1 \,\big\},$$

$$D'_1 = \big\{\, \lambda_{L'}(x) \;\big|\; x \in D_1 \setminus D_2 \,\big\}, D'_2 = \big\{\, \lambda_{L'}(x) \;\big|\; x \in D_2 \setminus D_1 \,\big\},$$

$$L''_1 = (L'_1 \setminus L'_2) \vee L_1, \quad L''_2 = (L'_2 \setminus L'_1) \vee L_2,$$

$$n'_1 = n_1 \wedge \bigwedge_{x\in E'_1} x \wedge \bigwedge_{x\in D'_1} \overline{x} \wedge \bigwedge_{(x,y)\in L''_1} x \Leftrightarrow y,$$

$$n'_2 = n_2 \wedge \bigwedge_{x\in E'_2} x \wedge \bigwedge_{x\in D'_2} \overline{x} \wedge \bigwedge_{(x,y)\in L''_2} x \Leftrightarrow y.$$

(recall that the $\vee$ operator on equivalence relations represents their intersection). If the two arguments were already normalized, then it is proven that the result of the application of this algorithm will be already normalized (otherwise a further call to the normalization operator is required).

4.2.5 Entailed, disentailed and equivalent variables

Operations that search and eliminate entailed, disentailed and equivalent variables are pretty straightforward, provided that the internal representation is already normalized: that's why these operations start with a call to the normalization algorithm. The operations do nothing more than returning and/or clearing the corresponding part of the internal representation, with the only exception coming from the fact that entailed and disentailed variables are all equivalent to each other.

4.2.6 Projections and variable renaming

Operations that rename variables can be easily implemented by renaming all components of the internal representation separately. Although renaming with respect to an equivalence relation may denormalize the internal representation, shift renaming of course does not.

For projections, the main problem is that we cannot project all components separately, as it is proven that this may alter the semantics for the case where we project away variables that appear in more than one component. Note that this case may also occur in a normalized representation whenever we project away a leader for L that also labels a node in n. Our solution is to start by normalizing the representation, then replacing each occurence of all of these leaders l in n with the minimum variable that is equivalent to l in L and is not going to be projected away. Afterwards we can project on each component separately.

5 Experimental evaluation

5.1 Methodology

Here we report the benchmark results obtained by testing China against all programs from the China Benchmark Suite, for which you can find more informations at **http://www.cs.unipr.it/China/Benchmarks/**. Each table shows the results obtained by using a specific internal representation and compares the performance of China's native implementation with the one obtained by integrating CORAL into China.

The numbers shown are the user run-times in seconds that are required to perform groundness analysis of each program. We discarded all programs for which the analysis always took less than a tenth of a second and marked all times greater than 300 seconds as TIMEOUTs; the indication of output differences is not significant if a TIMEOUT occurs. We also report whether there are differences between the results of analysis for the two implementations: this is almost always caused by the fact that `Too_Many_BDD_Nodes` is thrown in different contexts, which leads to different applications of upward approximation (the new implementation usually approximates more). All results were obtained on a AMD Phenom 8650 machine running in 32-bit mode, with 2 GB of RAM and running Linux 2.6.27.

Be aware that CORAL is still penalized by an incomplete integration with China and by a less efficient implementation of class `Bit_Set` that will be improved in the future.

5.2 Results

5.2.1 Plain ROBDD representation

Program	China	China + CORAL	Differs
8puzzle.pl	0.10	0.11	No
CoVer.pl	0.08	0.10	No
QG5-7_mg4.pl	0.10	0.10	No
XRayVer33.pl	7.67	7.47	No
aaaaa.pl	56.28	52.93	No
aircraft.pl	0.94	0.95	No
aleph.pl.unsound	200.20	198.06	No
ann.pl	0.30	0.30	No
aqua_c.pl	TIMEOUT	TIMEOUT	Yes
arch1.pl	0.42	0.42	No
back52.pl.unsound	243.20	232.36	No
bamspec.pl	94.35	64.18	No
barnes_hut.pl	0.21	0.20	No
bmtp.pl	16.06	15.58	No
botworld.pl	2.07	2.21	No
bryant.pl	0.60	0.67	No
caslog.pl	TIMEOUT	252.50	Yes
chartgraph.pl	4.17	3.97	No
chat80.pl	49.29	45.72	No
chat_parser.pl	2.95	3.01	No
chess.pl	0.66	0.68	No
chillin.pl	111.67	100.03	No
circan.pl	0.14	0.16	No
cobweb.pl	5.12	5.10	No
colan-all.pl.unsound	188.78	189.60	No
cselcomp.pl	229.45	126.03	No
curry2prolog.pl	27.70	26.55	No
diagnoser.pl.unsound	3.68	3.72	No
essln.pl.unsound	0.11	0.12	No
ezan.pl	3.40	3.40	No
fecht.pl	1.43	1.35	No
genlang-2.5.pl.unsound	15.82	15.20	No
glop.pl	0.41	0.40	No
gnup-1.1.0_pl2wam.pl	7.00	7.04	No
horatio.pl	TIMEOUT	TIMEOUT	No
horgen.pl	29.57	29.04	No
ileanTAP.pl	69.44	67.62	No
index.pl.unsound	0.58	0.60	No
indyv1.8.pl	0.45	0.44	No
k4_tp.pl	0.21	0.18	No
kilimanjaro-all.pl	103.00	108.63	No

Program	China	China + CORAL	Differs
kish_andi.pl	TIMEOUT	TIMEOUT	No
kore-ie.pl.unsound	96.98	87.12	No
ldl-all.pl	TIMEOUT	TIMEOUT	No
lemmatiz.pl	1.16	1.24	No
lg_sys.pl	TIMEOUT	TIMEOUT	No
lgt20.pl.unsound	2.41	2.55	No
lilp.pl	0.87	0.85	No
linTAP.pl	66.26	58.58	No
linger_old.pl.unsound	18.39	19.19	No
linus.pl.unsound	0.30	0.31	No
llprover.pl	0.51	0.51	No
lojban.pl	1.29	1.26	No
lptp-1.05.pl	1.45	1.38	No
lptp-1.06.pl	1.32	1.43	No
map.pl.unsound	6.12	6.12	No
markus.pl	245.71	203.42	No
mdgtools-1.0.pl	256.51	240.19	No
metutor.pl	0.25	0.25	No
mfoil.pl	25.09	24.51	No
mgpl4.pl	0.62	0.64	No
mgtp-g-I-temp.pl	0.14	0.14	No
mikev203.pl	0.10	0.11	No
mixtus-all.pl	62.71	57.97	No
mm.pl	0.07	0.08	No
motel.pl.unsound	TIMEOUT	TIMEOUT	No
mt-all.pl.unsound	222.32	231.19	No
music.pl	7.96	8.19	No
nand_wamcc.pl	103.05	99.75	No
newexpobdd.pl	29.20	29.94	No
nuc4-unfixed.pl	16.27	16.21	No
oldchina.pl	TIMEOUT	97.25	Yes
pappiall.pl	0.30	0.30	No
pappienglishall.pl	0.15	0.18	No
petsan.pl	34.09	32.85	No
peval.pl	0.08	0.08	No
piza-0.9.22.pl.unsound	206.84	191.84	No
pl2wam.pl	7.02	6.58	No
plaiclp.pl	64.50	63.83	No
pljava.pl	0.23	0.20	No
pmatch.pl	131.24	127.83	No
poker.pl	1.88	1.73	No
prism-1.1.pl.unsound	0.16	0.15	No
profit.pl.unsound	0.53	0.50	No
protein.pl.unsound	55.84	55.24	No

Program	China	China + CORAL	Differs
puzzle.pl	TIMEOUT	37.56	Yes
qdjanus.pl.unsound	1.20	1.21	No
qmlattice.pl.unsound	0.08	0.08	No
raytrace_inst.pl	0.10	0.08	No
reform_compiler.pl.unsound	TIMEOUT	TIMEOUT	Yes
reg.pl	3.50	3.33	No
rubik.pl	185.78	201.85	No
s4_tp.pl	0.21	0.19	No
salvini.pl.unsound	0.14	0.16	No
sax.pl	5.18	5.03	No
scc.pl	1.41	1.38	No
scc2.pl	0.76	0.69	No
semi.pl	0.22	0.23	No
semigroup.pl	29.69	28.68	No
sg_mc_big.pl	0.97	0.57	No
sim.pl	161.73	147.79	No
simple_analyzer.pl	2.32	2.28	No
slice-all.pl	0.13	0.12	No
sprftp.pl.unsound	8.99	8.59	No
spsys.pl	1.20	1.17	No
strips.pl	2.82	2.75	No
synth.pl	2.38	2.25	No
tictactoe.pl	2.12	2.01	No
tricia.pl	TIMEOUT	76.43	Yes
trs.pl	47.02	47.93	No
vhdl97_parser.pl	3.03	2.97	No

Here the performance shown by the new implementation is very often better; we suspect that the reason lies in an unoptimal use of `and_vs_true` made by the previous implementation.

5.2.2 ROBDD plus entailed variables representation

Program	China	China + CORAL	Differs
8puzzle.pl	0.11	0.15	No
CoVer.pl	0.09	0.11	No
QG5-7_mg4.pl	0.09	0.11	No
XRayVer33.pl	7.87	12.63	No
aaaaa.pl	77.50	81.01	Yes
aircraft.pl	0.16	0.80	No
aleph.pl.unsound	137.82	TIMEOUT	Yes
ann.pl	0.10	0.21	No
aqua_c.pl	TIMEOUT	TIMEOUT	Yes
arch1.pl	0.48	0.44	No
back52.pl.unsound	261.87	TIMEOUT	Yes
bamspec.pl	92.24	123.12	No
barnes_hut.pl	0.01	0.06	No
bmtp.pl	21.61	28.86	No
botworld.pl	1.51	4.03	No
bryant.pl	0.22	0.54	No
caslog.pl	215.46	TIMEOUT	Yes
chartgraph.pl	5.44	9.86	No
chat80.pl	53.82	53.69	Yes
chat_parser.pl	2.16	3.88	No
chess.pl	0.54	0.62	No
chillin.pl	106.51	100.45	Yes
circan.pl	0.23	0.20	No
cobweb.pl	1.90	4.14	No
colan-all.pl.unsound	175.89	256.15	Yes
cselcomp.pl	170.53	230.85	Yes
curry2prolog.pl	25.28	28.80	No
diagnoser.pl.unsound	3.64	3.96	No
essln.pl.unsound	0.08	0.13	No
ezan.pl	3.02	3.21	No
fecht.pl	1.81	1.67	No
genlang-2.5.pl.unsound	13.74	19.51	Yes
glop.pl	0.41	0.44	No
gnup-1.1.0_pl2wam.pl	7.43	1.17	No
horatio.pl	TIMEOUT	TIMEOUT	Yes
horgen.pl	39.11	42.60	No
ileanTAP.pl	73.10	70.52	No
index.pl.unsound	0.51	0.59	No
indyv1.8.pl	0.29	0.74	No
k4_tp.pl	0.20	0.20	No
kilimanjaro-all.pl	116.38	170.76	Yes

Program	China	China + CORAL	Differs
kish_andi.pl	TIMEOUT	TIMEOUT	Yes
kore-ie.pl.unsound	57.50	144.68	No
ldl-all.pl	133.08	172.73	Yes
lemmatiz.pl	1.42	1.60	No
lg_sys.pl	TIMEOUT	TIMEOUT	Yes
lgt20.pl.unsound	1.13	0.12	No
lilp.pl	0.67	0.04	No
linTAP.pl	64.28	72.00	No
linger_old.pl.unsound	17.75	22.09	No
linus.pl.unsound	0.20	0.28	No
llprover.pl	0.48	0.02	No
lojban.pl	0.47	1.26	No
lptp-1.05.pl	1.40	1.39	No
lptp-1.06.pl	1.42	0.06	No
map.pl.unsound	5.59	6.13	No
markus.pl	TIMEOUT	TIMEOUT	Yes
mdgtools-1.0.pl	TIMEOUT	284.69	Yes
metutor.pl	0.10	0.21	No
mfoil.pl	19.42	36.54	Yes
mgpl4.pl	0.45	0.56	No
mgtp-g-I-temp.pl	0.16	0.16	No
mikev203.pl	0.10	0.18	No
mixtus-all.pl	63.20	62.42	No
mm.pl	0.10	0.12	No
motel.pl.unsound	TIMEOUT	TIMEOUT	Yes
mt-all.pl.unsound	237.10	TIMEOUT	Yes
music.pl	6.63	8.48	No
nand_wamcc.pl	94.43	99.13	Yes
newexpobdd.pl	31.04	34.57	No
nuc4-unfixed.pl	19.21	16.29	No
oldchina.pl	145.02	230.63	Yes
pappiall.pl	0.24	0.02	No
pappienglishall.pl	0.12	0.17	No
petsan.pl	28.14	34.51	Yes
peval.pl	0.08	0.11	No
piza-0.9.22.pl.unsound	187.16	186.65	Yes
pl2wam.pl	7.06	6.90	No
pljava.pl	0.16	0.25	No
pmatch.pl	148.63	215.16	Yes
poker.pl	0.48	0.87	No
prism-1.1.pl.unsound	0.14	0.15	No
profit.pl.unsound	0.23	0.47	No
protein.pl.unsound	60.81	83.61	No
puzzle.pl	48.00	46.54	No

Program	China	China + CORAL	Differs
qdjanus.pl.unsound	0.84	1.03	No
qmlattice.pl.unsound	0.08	0.10	No
raytrace_inst.pl	0.06	0.02	No
reform_compiler.pl.unsound	TIMEOUT	TIMEOUT	Yes
reg.pl	2.78	2.67	No
rubik.pl	202.56	208.93	No
s4_tp.pl	0.19	0.21	No
salvini.pl.unsound	0.12	0.18	No
sax.pl	3.99	4.70	No
scc.pl	1.23	0.07	No
scc2.pl	0.74	0.74	No
semi.pl	0.01	0.00	No
semigroup.pl	11.56	19.96	No
sg_mc_big.pl	0.02	0.11	No
sim.pl	54.22	130.56	Yes
simple_analyzer.pl	2.07	2.11	No
slice-all.pl	0.10	0.13	No
sprftp.pl.unsound	2.96	7.60	No
spsys.pl	1.14	1.25	No
strips.pl	2.82	2.75	No
synth.pl	0.90	2.25	No
tictactoe.pl	0.60	2.18	No
tricia.pl	79.28	73.10	No
trs.pl	74.58	277.57	Yes
vhdl97_parser.pl	2.32	2.42	No

This representation is better than the previous one only on some examples. The performance of the new implementation is generally not too much worse, except for a few programs.

5.2.3 ROBDD plus entailed and equivalent variables representation

Program	China	China + CORAL	Differs
8puzzle.pl	0.06	0.07	Yes
CoVer.pl	0.06	0.09	No
QG5-7_mg4.pl	0.00	0.06	No
XRayVer33.pl	0.11	0.12	No
aaaaa.pl	0.66	0.78	Yes
aircraft.pl	0.05	0.00	No
aleph.pl.unsound	0.12	0.18	Yes
ann.pl	0.00	0.03	Yes
aqua_c.pl	0.03	0.96	Yes
arch1.pl	0.00	0.00	No
back52.pl.unsound	2.12	1.86	Yes
bamspec.pl	0.16	0.18	Yes
barnes_hut.pl	0.00	0.01	No
bmtp.pl	0.20	0.38	Yes
botworld.pl	0.01	0.00	No
bryant.pl	0.02	0.03	No
caslog.pl	0.37	0.63	Yes
chartgraph.pl	0.01	0.03	No
chat80.pl	0.28	0.37	Yes
chat_parser.pl	0.00	0.08	Yes
chess.pl	0.05	0.06	Yes
chillin.pl	0.16	0.23	Yes
circan.pl	7.80	3.11	Yes
cobweb.pl	0.04	0.05	Yes
colan-all.pl.unsound	0.18	0.23	Yes
cselcomp.pl	0.19	0.27	Yes
curry2prolog.pl	0.11	0.18	No
diagnoser.pl.unsound	0.14	0.26	Yes
essln.pl.unsound	0.05	0.07	Yes
ezan.pl	0.08	0.10	Yes
fecht.pl	0.01	0.01	No
genlang-2.5.pl.unsound	0.21	0.02	Yes
glop.pl	0.42	0.41	No
gnup-1.1.0_pl2wam.pl	0.08	0.00	Yes
horatio.pl	0.18	0.25	Yes
horgen.pl	0.08	0.11	No
ileanTAP.pl	0.08	0.07	No
index.pl.unsound	0.05	0.05	Yes
indyv1.8.pl	0.14	0.17	No
k4_tp.pl	0.18	0.16	No
kilimanjaro-all.pl	3.48	1.47	Yes

Program	China	China + CORAL	Differs
kish_andi.pl	0.33	0.38	Yes
kore-ie.pl.unsound	0.04	0.09	Yes
ldl-all.pl	0.70	1.00	Yes
lemmatiz.pl	1.20	1.64	No
lg_sys.pl	0.71	0.75	Yes
lgt20.pl.unsound	0.02	0.00	No
lilp.pl	0.02	0.00	Yes
linTAP.pl	0.05	0.05	Yes
linger_old.pl.unsound	0.00	0.07	Yes
linus.pl.unsound	0.00	0.00	No
llprover.pl	0.03	0.06	Yes
lojban.pl	0.04	0.08	No
lptp-1.05.pl	0.17	0.28	Yes
lptp-1.06.pl	0.17	0.26	Yes
map.pl.unsound	0.22	0.02	Yes
markus.pl	0.20	0.57	Yes
mdgtools-1.0.pl	0.47	0.02	Yes
metutor.pl	0.03	0.00	No
mfoil.pl	0.02	0.00	Yes
mgpl4.pl	0.00	0.09	Yes
mgtp-g-I-temp.pl	0.04	0.00	No
mikev203.pl	0.00	0.00	No
mixtus-all.pl	0.12	0.24	Yes
mm.pl	0.00	0.00	No
motel.pl.unsound	TIMEOUT	0.10	Yes
mt-all.pl.unsound	4.48	2.18	Yes
music.pl	0.52	0.04	No
nand_wamcc.pl	0.18	0.17	Yes
newexpobdd.pl	2.66	2.76	No
nuc4-unfixed.pl	0.06	0.09	Yes
oldchina.pl	0.21	0.12	Yes
pappiall.pl	0.19	0.22	No
pappienglishall.pl	0.00	0.00	No
petsan.pl	1.32	4.42	Yes
peval.pl	0.06	0.08	No
piza-0.9.22.pl.unsound	0.61	0.05	Yes
pl2wam.pl	0.00	0.14	Yes
plaiclp.pl	0.11	0.19	Yes
pljava.pl	0.06	0.11	No
pmatch.pl	0.06	0.06	Yes
poker.pl	0.04	0.08	No
prism-1.1.pl.unsound	0.00	0.00	No
profit.pl.unsound	0.05	0.10	No
protein.pl.unsound	0.54	0.35	Yes

Program	China	China + CORAL	Differs
puzzle.pl	9.43	12.55	Yes
qdjanus.pl.unsound	0.10	0.16	Yes
qmlattice.pl.unsound	0.06	0.06	No
raytrace_inst.pl	0.02	0.03	No
reform_compiler.pl.unsound	4.86	3.61	Yes
reg.pl	0.06	0.10	Yes
rubik.pl	4.19	8.97	No
s4_tp.pl	0.20	0.17	No
salvini.pl.unsound	0.04	0.06	Yes
sax.pl	0.31	0.39	Yes
scc.pl	0.07	0.04	Yes
scc2.pl	0.03	0.03	Yes
semi.pl	0.00	0.00	No
semigroup.pl	0.03	0.09	No
sg_mc_big.pl	0.01	0.03	No
sim.pl	0.00	0.00	Yes
simple_analyzer.pl	0.03	0.04	Yes
slice-all.pl	0.08	0.12	No
sprftp.pl.unsound	0.03	0.06	Yes
spsys.pl	0.09	0.10	Yes
strips.pl	0.06	0.06	Yes
synth.pl	0.16	0.18	Yes
tictactoe.pl	0.14	0.16	Yes
tricia.pl	12.67	28.16	Yes
trs.pl	0.10	0.11	Yes
vhdl97_parser.pl	0.46	0.54	Yes

This representation performs much better than the previous ones, although the new implementation is generally less efficient and approximates more.

6 Conclusions

6.1 Future developments

The efficiency of CORAL, and in particular of the portion of China that utilizes CORAL, must still be improved. While the results of the experimental evaluation were decent, we still haven't been able to match the efficiency and the precision of the native China implementation: we are currently able to match or beat the native implementation only by applying upward approximation more intensively.

Instead of using a single Boolean data member to record whether each representation $\langle n, E, D, L\rangle$ is normalized or not, we may also consider recording all states of partial normalization. For example, we may use multiple Boolean data members to record:

Whether we know that there are no entailed variables in n or not.

Whether we know that all variables in E do not occur in n or not.

Whether we know that there are no disentailed variables in n or not.

Whether we know that all variables in D do not occur in n or not.

... (etc.)

These informations may allow us to avoid unnecessary operations in various algorithms. Both CORAL and China still need extensive debugging before release.

Bibliography

[Bry86] R. E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, 1986.

[Bry92] R. E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, 1992.

[BS98] R. Bagnara and P. Schachte. Efficient implementation of *Pos*. Technical Report 98/5, Department of Computer Science, The University of Melbourne, Australia, 1998.

[MS93] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. *ACM Letters on Programming Languages and Systems*, 2(1–4):181–196, 1993.