Contents

3 Symbolic Graph Representation 1
  3.1 Symbolic Invariant Verification 1
    3.1.1 Symbolic Search 2
    3.1.2 Symbolic Implementation 4
  3.2 Binary Decision Diagrams 7
    3.2.1 Ordered Binary Decision Graphs 8
    3.2.2 Ordered Binary Decision Diagrams 10
    3.2.3 Operations on BDDs 14
    3.2.4 Symbolic Search using BDDs 18

Chapter 3

Symbolic Graph Representation

3.1 Symbolic Invariant Verification

As the invariant-verification problem is Pspace-hard, we cannot hope to find a polynomial-time solution. There are, however, heuristic that perform well on many instances of the invariant-verification problem that occur in practice. One such heuristic is based on a symbolic reachability analysis of the underlying transition graph. Symbolic graph algorithms operate on implicit (or symbolic) —rather than explicit (or enumerative)— representations of regions. While an enumerative representation of the region σ is a list of the states in σ, a symbolic representation of σ is a set of constraints that identifies the states in σ. For example, for an integer variable x, the constraint $20 \leq x \leq 99$ identifies the set $\{20, 21, \ldots, 99\}$ of 80 states. A symbolic region representation may be much more succinct than the corresponding enumerative representation.

3.1.1 Symbolic Search

Consider a transition graph G with the initial region σ^I. Since the reachable region σ^R equals $(\cup i \in \mathbb{N}.\, post^i(\sigma^I))$, it can be computed from σ^I by iterating the function *post* on regions. This observation leads to a symbolic algorithm for solving the reachability problem. Unlike enumerative algorithms, the symbolic algorithm does not need to test membership of a state in a region, nor does it require to enumerate all states in a region.

We use the abstract type symreg, called symbolic region, to represent regions. The abstract type symreg supports five operations.

$\cup$: **symreg** $\times$ **symreg** $\mapsto$ **symreg**. The operation $\sigma \cup \tau$ returns the union of the regions σ and τ.

$\cap$: **symreg** $\times$ **symreg** $\mapsto$ **symreg**. The operation $\sigma \cap \tau$ returns the intersection of the regions σ and τ.

$=$: **symreg** $\times$ **symreg** $\mapsto \mathbb{B}$. The operation $\sigma = \tau$ returns *true* iff the regions σ and τ contain the same states.

$\subseteq$: **symreg** $\times$ **symreg** $\mapsto \mathbb{B}$. The region $\sigma \subseteq \tau$ returns *true* iff every state in σ is contained in τ.

EmptySet: **symreg**. The empty set of states.

Since $\sigma \subseteq \tau$ iff $\sigma \cup \tau = \tau$, the inclusion test can be implemented using the union and equality test. Alternatively, the equality test can be implemented using two inclusion tests: $\sigma = \tau$ iff both $\sigma \subseteq \tau$ and $\tau \subseteq \sigma$.

The abstract type of transition graphs is changed to symgraph, called symbolic graph, which supports two operations.

InitReg: **symgraph** $\mapsto$ **symreg**. The operation *InitReg*(G) returns the initial region of G.

PostReg: **symreg** $\times$ **symgraph** $\mapsto$ **symreg**. The operation *PostReg*(σ, G) returns the successor region $post_G(\sigma)$.

Algorithm 3.1 searches the input graph in a breadth-first fashion using symbolic types for the input graph and for regions. After j iterations of the repeat loop, the set σ^R equals $post^{\leq j}(\sigma^I)$. This is depicted pictorially in Figure 3.2.

Theorem 3.1 [Symbolic graph search] Let G be a transition graph, and let σ^T be a region of G. Algorithm 3.1, if it terminates, correctly solves the reachability problem (G, σ^T). Furthermore, if there exists $j \in \mathbb{N}$ such that (1) every reachable state is the sink of some initialized trajectory of length at most j (i.e. $\sigma^R = (\cup i \leq j.\, post^i(\sigma^I))$), or (2) some state in the target region σ^T is the sink of some initialized trajectory of length j (i.e. $\sigma^T \cap post^j(\sigma^I)$ is nonempty), then the algorithm terminates within j iterations of the repeat loop.

Algorithm 3.1 [Symbolic Search]

Input: a transition graph G, and a region σ^T of G.
Output: the answer to the reachability problem (G, σ^T).

input G: symgraph; σ^T: symreg;
local σ^R: symreg;
begin
  $\sigma^R := \mathit{InitReg}(G)$;
  repeat
    if $\sigma^R \cap \sigma^T \neq \mathit{EmptySet}$ then return Yes fi;
    if $\mathit{PostReg}(\sigma^R, G) \subseteq \sigma^R$ then return No fi;
    $\sigma^R := \sigma^R \cup \mathit{PostReg}(\sigma^R, G)$
    forever
  end.

Figure 3.1: Symbolic search

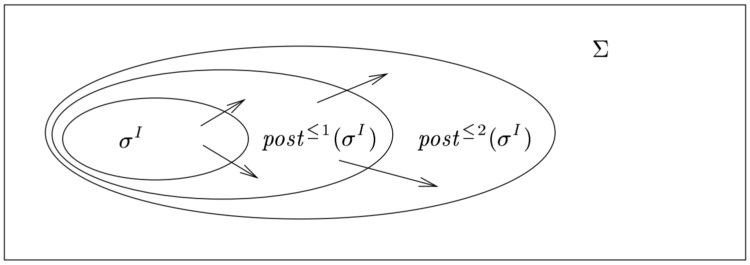


Figure 3.2: Symbolic computation of the reachable region

In particular, if the input graph G is finite with n states, then Algorithm 3.1 terminates within n iterations of the repeat loop.

Exercise 3.1 {T3} [Fixpoint view of breadth-first search] Let $G = (\Sigma, \sigma^I, \rightarrow)$ be a transition graph. The subset relation $\subseteq$ is a complete partial order on the set $\mathbf{2}^\Sigma$ of regions of G. Let f be a function from $\mathbf{2}^\Sigma$ to $\mathbf{2}^\Sigma$ such that for each region $\sigma \subseteq \Sigma$,

$$f(\sigma) \ = \ \sigma^I \cup post_G(\sigma).$$

(1) Prove that the function f is monotonic, $\bigcup$-continuous, and $\bigcap$-continuous. (2) What is the least fixpoint μf, and what is the greatest fixpoint νf? Conclude that Algorithm 3.1 can be viewed as a computation of the least fixpoint μf by successive approximation. ■

Exercise 3.2 {T2} [Enumerative region operations] Suppose we implement the abstract type symreg as a queue of states. Write algorithms that implement all boolean operations, emptiness test, equality test, and inclusion test. What is the cost of each operation, and what is the total cost of Algorithm 3.1? Repeat the exercise assuming that the type symreg is implemented as a boolean array indexed by states. ■

Exercise 3.3 {P2} [Witness reporting in symbolic search] Write an algorithm for symbolic search that, given an input transition graph G and a region σ^T of G, outputs Done, if the reachability problem (G, σ^T) has the answer No; and a witness for the reachability problem (G, σ^T), otherwise. Assume that the following two additional operations are supported by our abstract types.

Element: **symreg** $\mapsto$ **state**. The operation $Element(\sigma)$ returns a state belonging to σ.

PreReg: **symreg** $\times$ **symgraph** $\mapsto$ **symreg**. The operation $PreReg(\sigma, G)$ returns the predecessor region $pre_G(\sigma)$.

■

3.1.2 Symbolic Implementation

Consider the transition graph over the state space Σ_X for a finite set X of typed variables. Let the type of a variable x be denoted by $\mathbb{T}_x$. The type $\mathbb{T}_X$ denotes the product type $\Pi_{x \in X} \mathbb{T}_x$. Then, the type **state** is the product type $\mathbb{T}_X$. The type symreg is parametrized by the state type **state**, and we write **symreg**[**state**]. A transition is a pair of states, and thus, has type $\mathbb{T}_X \times \mathbb{T}_X$. Equivalently, a transition can be viewed as a valuation for the set $X \cup X'$, where the values of the unprimed variables specify the source state of the transition and the values of the primed variables specify the sink state of the transition. Consequently, the type of a transition is $\mathbb{T}_{X \cup X'}$. Then, the symbolic representation of the transition graph G with the state space Σ_X is a record $\{G\}_s$ with two components, (1) the initial region $\{\sigma^I\}_s$ of type **symreg**$[\mathbb{T}_X]$ and (2) the transition relation $\{\rightarrow\}_s$ of type **symreg**$[\mathbb{T}_{X \cup X'}]$.

We consider the operations renaming and existential-quantifier elimination on the abstract type **symreg**$[\mathbb{T}_X]$.

Rename: **variable** $\times$ **variable** $\times$ **symreg**$[\mathbb{T}_X] \mapsto$ **symreg**$[\mathbb{T}_{X[x:=y]}]$. For variables x and y of the same type, the operation $Rename(x, y, \sigma)$ returns the renamed region $\sigma[x := y]$.

Exists: **variable** $\times$ **symreg**$[\mathbb{T}_X] \mapsto$ **symreg**$[\mathbb{T}_{X \setminus \{x\}}]$. The operation $Exists(x, \sigma)$ returns the region $\{s \in \Sigma_{X \setminus \{x\}} \mid (\exists m.\, s[x := m] \in \sigma)\}$.

The operations renaming and existential-quantifier elimination naturally extend to variable sets. For variable sets $X = \{x_1, \ldots x_n\}$ and $Y = \{y_1, \ldots y_n\}$ such that, for all $1 \leq i \leq n$, the variables x_i and y_i are of the same type, we write $\mathit{Rename}(X, Y, \sigma)$ for $\mathit{Rename}(x_n, y_n, \mathit{Rename}(\ldots \mathit{Rename}(x_1, y_1, \sigma)))$. Similarly, for a variable set $X = \{x_1, \ldots x_n\}$, we write $\mathit{Exists}(X, \sigma)$ for $\mathit{Exists}(x_n, \mathit{Exists}(\ldots \mathit{Exists}(x_1, \sigma)))$. We implicitly use simple forms of type inheritence and type polymorphism. For instance, if the set X of variables is a subset of Y then a region of type $\mathbb{T}_X$ is also a region of type $\mathbb{T}_Y$; if the region σ is of type $\mathbb{T}_X$ and the region τ is of type $\mathbb{T}_Y$ then the intersection $\sigma \cap \tau$ has type $\mathbb{T}_{X \cup Y}$.

Consider the symbolic representation $(\{\sigma^I\}_s, \{\rightarrow\}_s)$ of a transition graph G, and a region σ. Then, to compute a representation of the region $\mathit{post}_G(\sigma)$, we can proceed as follows. First, we conjunct σ with $\{\rightarrow\}_s$ to obtain the set of transitions originating in σ. Second, we project the result onto the set X' of variables by eliminating the variables in X. This yields a representation of the successor region $\mathit{post}_G(\sigma)$ in terms of the primed variables. Renaming each primed variable x' to x, then, leads to the desired result. In summary, the operation *PostReg* can be implemented using existential-quantifier elimination and renaming:

$$\mathit{PostReg}(\sigma, \{G\}_s) \;=\; \mathit{Rename}(X', X, \mathit{Exists}(X, \sigma \cap \{\rightarrow\}_s))$$

A natural choice for a symbolic representation of regions is boolean expressions. An expression is usually represented by its parse tree, or by a directed acyclic graph that allows sharing of common subexpressions to avoid duplication of syntactically identical subexpressions. If X contains only propositions, then we can represent a region as a propositional formula. The operation $\cup$ corresponds to disjunction of formulas, and the operation $\cap$ corresponds to conjunction of formulas. Both operations can be performed in constant time. The constant *EmptySet* corresponds to the formula *false*. Renaming corresponds to textual substitution, and can be performed in constant time. Existential-quantifier elimination can be performed in linear time:

$$\mathit{Exists}(x, p) \;=\; (p[x := \mathit{true}] \vee p[x := \mathit{false}]).$$

The satisfiability problem for propositional formulas is NP-complete and, therefore, the validity problem and the equivalence problem for propositional formulas are coNP-complete. The equality test corresponds to checking equivalence, and the inclusion test corresponds to checking validity of the implication. Thus, both these operations are coNP-complete.

The representation of regions as propositional formulas is possible for the propositional invariant-verification problem. Given a propositional module P with the set X of variables, the symbolic representation of the transition graph G_P consists of (1) [the initial predicate] a propositional formula $\{\sigma^I\}_s = q^I$ over X, and (2) [the transition predicate] a propositional formula $\{\rightarrow\}_s = q^T$ over $X \cup X'$.

The lengths of both formulas are linear in the size of the module description. The initial predicate is obtained by taking conjunction of the initial commands of all the atoms, and the transition predicate is obtained by taking conjunction of the update commands of all the atoms.

Remark 3.1 [Module operations for formula representation] The parallel composition of modules corresponds to the conjunction of initial and transition predicates, and the renaming of modules corresponds to the renaming of initial and transition predicates. The hiding of module variables does not affect the initial and transition predicates. ■

Analogously, enumerated formulas can be used as a symbolic representation of enumerated modules. Such a symbolic representation is linear in the size of the enumerated module. The complexities of implementing various operations on regions represented as enumerated formulas are analogous to the corresponding complexities for propositional formulas. In particular, union, intersection, renaming, and existential-quantifier elimination are easy, but equality and inclusion tests are hard, namely, coNP-complete.

Example 3.1 [Mutual exclusion] Recall Peterson's mutual-exclusion protocol from Chapter 1. A symbolic representation of the transition graph G_{P_1} has the set $\{pc_1, pc_2, x_1, x_2\}$ of variables, the initial predicate q_1^I:

$$pc_1 = outC, \tag{q_1^I}$$

and the transition predicate q_1^T:

$$\begin{array}{l}
\vee\, (pc_1 = outC \ \wedge\ pc_1' = reqC \ \wedge\ x_1' = x_2) \\
\vee\, (pc_1 = reqC \ \wedge\ (pc_2 = outC \ \vee\ x_1 \neq x_2) \ \wedge\ pc_1' = inC \ \wedge\ x_1' = x_1) \\
\vee\, (pc_1 = inC \ \wedge\ pc_1' = outC \ \wedge\ x_1' = x_1) \\
\vee\, (pc_1' = pc_1 \ \wedge\ x_1' = x_1).
\end{array}$$

Given a propositional formula p, $PostReg(p)$ corresponds to

$$Rename(\{pc_1', pc_2', x_1', x_2'\}, \{pc_1, pc_2, x_1, x_2\}, Exists(\{pc_1, pc_2, x_1, x_2\}, p \cap q_1^T)).$$

Consider the computation of $PostReg(q_1^I)$. First, we take the conjunction of q_1^I and q_1^T. The resulting formula can be rewritten after simplification as

$$\begin{array}{l}
\vee\, (pc_1 = outC \ \wedge\ pc_1' = reqC \ \wedge\ x_1' = x_2) \\
\vee\, (pc_1 = outC \ \wedge\ pc_1' = outC \ \wedge\ x_1' = x_1).
\end{array}$$

After eliminating the variables pc_1, x_1, and x_2, we obtain

$$pc_1' = outC \ \vee\ pc_1' = reqC.$$

Finally, renaming the primed variables to unprimed ones, yields the expression

$$pc_1 = outC \ \vee\ pc_1 = reqC$$

whcih captures the set of states reachable from the initial states in one round.
■

Exercise 3.4 {P2} [Mutual exclusion] Give the initial predicate and the transition predicate for Peterson's protocol $P_1 \| P_2$. Simulate Algorithm 3.1 for checking that Peterson's protocol satisfies the mutual-exclusion requirement. For each iteration of the repeat loop, give the state predicate $\{\sigma^R\}_s$ after quantifier elimination and simplification. ■

Recall the definition of a latch-reduced transition graph of a module from Chapter 2. The initial and transition predicates of the reduced graph can be obtained from the corresponding predicates of the original graph using existential-quantifier elimination. Let P be a module with latched variables $latchX_P$, initial predicate q^I, and transition predicate q^T. Then, the initial region of the reduced transition graph G^L_P equals $Exists(X_P \backslash latchX_P, q^I)$, and the transition predicate of G^L_P equals $Exists((X_P \cup X'_P) \backslash (latchX_P \cup latchX'_P), q^T)$.

Exercise 3.5 {P2} [Message passing] Give the initial predicate and the transition predicate for the reduced transition graph of the send-receive protocol SyncMsg from Chapter 1. ■

Exercise 3.6 {P3} [Propositional invariant verification] (1) Write algorithms that implement the type symreg as propositional formulas supporting the operations $\cup$, $\cap$, $=$, *EmptySet*, $\subseteq$, *Rename*, and *Exists*. (2) Write an algorithm that, given a propositional module P, constructs the symbolic representation of the transition graph G_P. The size of the symbolic graph representation should be within a constant factor of the size of the module description. ■

Exercise 3.7 {P3} [Enumerated invariant verification] Write a symbolic algorithm for solving the enumerated invariant-verification problem. The size of the symbolic graph representation should be within a constant factor of the size of the module description. ■

Exercise 3.8 {T3} [Backward search] (1) Develop a symmetric version of Algorithm 3.1 that iterates the operator *pre* starting with the target region σ^T. Which region operations are used by your algorithm? (2) Given a symbolic representation $\{\sigma\}_s$ of the region σ, define a symbolic representation of the region $pre(\sigma)$ (use only positive boolean operations and quantifier elimination). ■

3.2 Binary Decision Diagrams

Binary decision diagrams (BDDs) provide a compact and canonical representation for propositional formulas (or, equivalently, for boolean functions). The BDD-representation of propositional formulas is best understood by first considering a related structure called an ordered binary decision graph (BDG).

3.2.1 Ordered Binary Decision Graphs

Let X be a set containing k boolean variables. A boolean expression p over X represents a function from $\mathbb{B}^k$ to $\mathbb{B}$. For a variable x in X, the following equivalence, called the Shannon expansion of p around the variable x, holds:

$$p \equiv (\neg x \wedge p[x := \mathit{false}]) \vee (x \wedge p[x := \mathit{true}]).$$

Since the boolean expressions $p[x := \mathit{true}]$ and $p[x := \mathit{false}]$ are boolean functions with domain $\mathbb{B}^{k-1}$, the Shannon expansion can be used to recursively simplify a boolean function. This suggests representing boolean functions as decision graphs.

A decision graph is a directed acyclic graph with two types of vertices, terminal vertices and internal vertices. The terminal vertices have no outgoing edges, and are labeled with one of the boolean constants. Each internal vertex is labeled with a variable in X, and has two outgoing edges, a left edge and a right edge. Every path from an internal vertex to a terminal vertex contains, for each variable x, at most one vertex labeled with x. Each vertex v represents a boolean function $r(v)$. Given an assignment s of boolean values to all the variables in X, the value of the boolean function $r(v)$ is obtained by traversing a path starting from v as follows. Consider an internal vertex w labeled with x. If $s(x)$ is 0, we choose the left-successor; if $s(x)$ is 1, we choose the right-successor. If the path terminates in a terminal vertex labeled with 0, the value $s(r(v))$ is 0; if the path terminates in a terminal vertex labeled with 1, the value $s(r(v))$ is 1.

Ordered decision graphs are decision graphs in which we choose a linear order $\prec$ over X, and require that the labels of internal vertices appear in an order that is consistent with $\prec$.

> Ordered Binary Decision Graph
>
> Let X be a finite set of propositions, and $\prec$ be a total order over X. An ordered binary decision graph B over $(X, \prec)$ consists of (1) [Vertices] a finite set V of vertices that is partitioned into two sets; internal vertices V^N and terminal vertices V^T, (2) [Root] a root vertex v^I in V, (3) [Labeling] a labeling function $\mathit{label} : V \mapsto X \cup \mathbb{B}$ that labels each internal vertex with a variable in X, and each terminal vertex with a constant in $\mathbb{B}$, (4) [Left edges] a left-child function $\mathit{left} : V^N \mapsto V$ that maps each internal vertex v to a vertex $\mathit{left}(v)$ such that if $\mathit{left}(v)$ is an internal vertex then $\mathit{label}(v) \prec \mathit{label}(\mathit{left}(v))$, and (5) [Right edges] a right-child function $\mathit{right} : V^N \mapsto V$ that maps each internal vertex v to a vertex $\mathit{right}(v)$ such that if $\mathit{right}(v)$ is an internal vertex then $\mathit{label}(v) \prec \mathit{label}(\mathit{right}(v))$.

The requirement that the labels of the children are greater than the label of a vertex ensures that every BDG is a finite and acyclic. Note that there is no requirement that every variable should appear as a vertex label along a path from

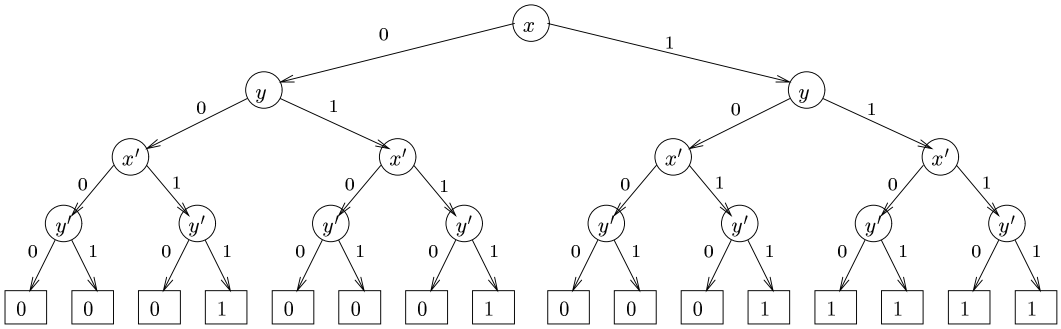


Figure 3.3: Ordered Binary Tree for $(x \wedge y) \vee (x' \wedge y')$

the root to a terminal vertex, but simply that the sequence of vertex labels along a path from the root to a terminal vertex is monotonically increasing according to $\prec$. The semantics of BDGs is defined by associating boolean expressions with the vertices.

Boolean Function of a BDG

Given a BDG B over $(X, \prec)$, let r be a function that associates each element of V with a boolean function over X such that $r(v)$ equals $label(v)$ if v is a terminal vertex, and equals

$$(\neg label(v) \wedge r(left(v))) \vee (label(v) \wedge r(right(v)))$$

otherwise. Define $r(B) = r(v^I)$ for the root v^I.

Example 3.2 [Binary decision graphs] A boolean constant is represented by a BDG that contains a single terminal vertex labeled with that constant. Figure 3.3 shows one possible BDG for the expression $(x \wedge y) \vee (x' \wedge y')$ with the ordering $x \prec y \prec x' \prec y'$. The left-edges are labeled with 0, and the right-edges are labeled with 1. The BDG of Figure 3.3 is, in fact, a tree. Figure 3.4 shows a more compact BDG for the same expression with the same ordering of variables. ■

Exercise 3.9 {T3} [Satisfying assignments] Write an algorithm that, given a BDG B over $(X, \prec)$, outputs an assignment s to X such that s satisfies $r(B)$. Write an algorithm that, given a BDG B over $(X, \prec)$, outputs the number of distinct assignments s to X such that s satisfies $r(B)$. What are the time complexities of your algorithms? ■

Two BDGs B and C are isomorphic if the corresponding labeled graphs are isomorphic. Two BDGs B and C are equivalent if the boolean expressions $r(B)$

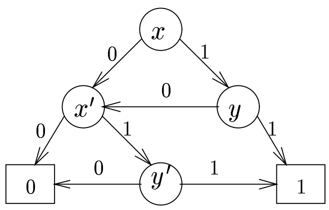


Figure 3.4: Ordered Binary Decision Diagram for $(x \wedge y) \vee (x' \wedge y')$

and $r(C)$ are equivalent. If B is a BDG over $(X, \prec)$, and v is a vertex of B, then the subgraph rooted at v is also a BDG over $(X, \prec)$. Two vertices v and w of the BDG B are isomorphic, if the subgraphs rooted at v and w are isomorphic. Similarly, two vertices v and w are equivalent, if the subgraphs rooted at v and w are equivalent.

Example 3.3 [Isomorphic and equivalent BDGs] The binary decision graphs of Figures 3.3 and 3.4 are not isomorphic, but are equivalent. In Figure 3.3, the subgraph rooted at vertex v_3 is a BDG that represents the boolean expression $x' \wedge y'$. The subgraphs rooted at vertices v_3, v_4, and v_5, are isomorphic. On the other hand, the vertices v_5 and v_6 are not isomorphic to each other. ■

Remark 3.2 [Isomorphism and Equivalence of BDGs] Let B and C be two BDGs over a totally ordered set $(X, \prec)$. Checking whether B and C are isomorphic can be performed in time linear in the number of vertices in B. Isomorphic BDGs are equivalent. However, isomorphism is not necessary for equivalence, as evidenced by the two nonisomorphic, but equivalent, BDGs of Figures 3.3 and 3.4. ■

3.2.2 Ordered Binary Decision Diagrams

An ordered binary decision diagram (BDD) is obtained from a BDG by applying the following two steps:

1. Identify isomorphic subgraphs.

2. Eliminate internal vertices with identical left and right successors.

Each step reduces the number of vertices while preserving equivalence. For instance, consider the BDG of Figure 3.3. Since vertices v_3 and v_4 are isomorphic, we can delete one of them, say v_4, and redirect the right-edge of the vertex v_1 to v_3. Now, since both edges of the vertex v_1 point to v_3, we can delete the vertex v_1 redirecting the left-edge of the root v_0 to v_3. Continuing in this manner, we obtain the BDD of Figure 3.4. It turns out that the above transformations are sufficient to obtain a canonical form.

Ordered Binary Decision Diagram
An ordered binary decision diagram over a totally ordered set $(X, \prec)$ is an ordered binary decision graph B over $(X, \prec)$ with vertices V and root v^I such that (1) [No isomorphic subgraphs] if v and w are two distinct vertices in V, then v is not isomorphic to w, and (2) [No redundancy] for every internal vertex v, the two successors $\mathit{left}(v)$ and $\mathit{right}(v)$ are distinct.

The next two proposition assert the basic facts about representing boolean expressions using BDDs: every boolean function has a unique, upto isomorphism, representation as a BDD.

Proposition 3.1 [Existence of BDDs] If p is a boolean expression over the set X of propositions and $\prec$ is a total order over X then there is a BDD B over $(X, \prec)$ such that $r(B)$ and p are equivalent.

Proposition 3.2 [Canonicity of BDDs] Let B and C be two BDDs over an ordered set $(X, \prec)$. Then, B and C are equivalent iff they are isomorphic.

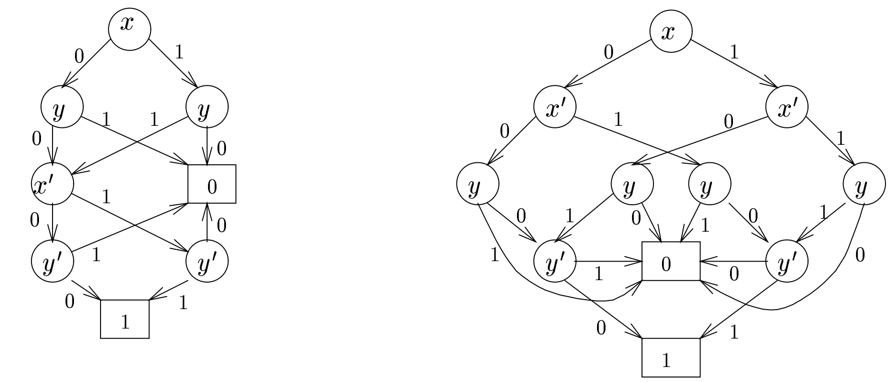
Exercise 3.10 {T5} [Existence and canonicity] Prove Proposition 3.1 and Proposition 3.2. ■

For a boolean function p and ordering $\prec$ of variables, let $B_{p,\prec}$ be the unique BDD B over $(X, \prec)$ such that $r(B)$ and p are equivalent.

Remark 3.3 [Checking Equivalence, Satisfiability, and Validity] Checking equivalence of two BDDs, with the same variable ordering, corresponds to checking isomorphism, and hence, can be performed in time linear in the number of vertices. The boolean constant 0 is represented by a BDD with a single terminal vertex labeled with 0, and the boolean constant 1 is represented by a BDD with a single terminal vertex labeled with 1. A boolean expression represented by the BDD B is satisfiable iff the root of B is not a terminal vertex labeled with 0. A boolean expression represented by the BDD B is valid iff the root of B is a terminal vertex labeled with 1. Thus, checking satisfiability or validity of boolean expressions is particularly easy, if we use BDD representation. Contrast this with representation as propositional formulas, where satisfiability is NP-complete and validity is coNP-complete. ■

The BDD of a boolean expression has the least number of vertices among all BDGs for the same expression using the same ordering.

Proposition 3.3 [Minimality of BDDs] Let B be an BDD over an ordered set $(X, \prec)$. If C is a BDG over $(X, \prec)$ and is equivalent to B, then C contains at least as many vertices as B.



Ordering: $x \prec y \prec x' \prec y'$ Ordering: $x \prec x' \prec y \prec y'$

Figure 3.5: Two BDDs for $(x \leftrightarrow y) \wedge (x' \leftrightarrow y')$.

Exercise 3.11 {T2} [Support sets] Let p be a boolean function over variables X. The support-set of p contains those variables x in X for which the boolean functions $p[x := \mathit{true}]$ and $p[x := \mathit{false}]$ are not equivalent. Show that a variable x belongs to the support-set of p iff no vertex in the BDD $B_{p,\prec}$ is labeled with x. ■

The size of the BDD may be exponential in the number of variables. Furthermore, one ordering may result in a BDD whose size is linear in the number of variables, while another ordering may result in a BDD whose size is exponential in the number of variables.

Example 3.4 [Variable ordering and BDD size] The size of the BDD representing a given predicate depends on the choice of the ordering of variables. Consider the predicate $(x \leftrightarrow y) \wedge (x' \leftrightarrow y')$. Figure 3.5 shows two BDDs for two different orderings. ■

Exercise 3.12 {T2} [Representation using BDDs] Consider the boolean expression

$$(x_1 \wedge x_2 \wedge x_3) \vee (\neg x_2 \wedge x_4) \vee (\neg x_3 \wedge x_4)$$

Choose a variable ordering for the variables $\{x_1, x_2, x_3, x_4\}$, and draw the resulting BDD. Can you reduce the size of the BDD by reordering the variables? ■

Exercise 3.13 {T3} [Exponential dependence on variable-ordering] Consider the set $X = \{x_1, x_2, \ldots x_{2k}\}$ with $2k$ variables. Consider the boolean expression

$$p\colon (x_1 \wedge x_2) \vee (x_3 \wedge x_4) \vee \cdots \vee (x_{2k-1} \wedge x_{2k}).$$

Show that (1) for the ordering $x_1 \prec x_2 \prec \cdots \prec x_{2k}$ the resulting BDD has $2k+2$ vertices, and (2) for the ordering $x_1 \prec x_{k+1} \prec x_2 \prec x_{k+2} \prec \cdots \prec x_k \prec x_{2k}$, the resulting BDD has 2^{k+1} vertices. ■

Given a boolean expression p over X, the linear order $\prec$ over X is optimal for p if, for every linear order $\prec'$ over X, $B_{p,\prec'}$ has at least as many vertices as $B_{p,\prec}$. Choosing an optimal ordering can lead to exponential saving, however, computing the optimal oredring itself is computationally hard.

Proposition 3.4 [Complexity of optimal ordering] The problem of checking, given a BDD B over $(X, \prec)$, whether the ordering $\prec$ is optimal for $r(B)$, is coNP-complete.

There are boolean functions whose BDD representation does not depend on the chosen ordering, and the BDD representation of some functions is exponential in the number of variables, irrespective of the ordering.

Example 3.5 [BDD for parity] Let X be a set of propositions. Consider the parity function *Parity*: for an assignment s, $s(\mathit{Parity}) = 1$ if the number of variables x with $s(x) = 1$ is even, and $s(\mathit{Parity}) = 0$ if the number of variables x with $s(x) = 1$ is odd. If X contains k variables, then irrespective of the chosen ordering $\prec$, $B_{\mathit{Parity},\prec}$ contains $2k + 1$ vertices. ■

Exercise 3.14 {T3} [BDD for addition] Let X be the set $\{x_0, x_1, y_0, y_1, \mathit{out}_0, \mathit{out}_1, \mathit{carry}\}$. Choose an appropriate ordering of the variables, and construct the BDD for the requirement that the output $\mathit{out}_1 \mathit{out}_0$, together with the carry bit carry, is the sum of the inputs $x_1 x_0$ and $y_1 y_0$. Is your choice of ordering optimal? ■

Exercise 3.15 {T5} [BDD for multiplication] Let X contain $2k$ variables $\{x_0, \ldots, x_{k-1}, y_0, \ldots, y_{k-1}\}$. For $0 \leq i < 2k$, let Mult_i denote the boolean function that denotes the i-th bit of the product of the two k-bit inputs, one encoded by the bits x_j and another encoded by the bits y_j. Prove that, for every ordering $\prec$ of the variables in X, there exists an index $0 \leq i < 2k$ such that the BDD $B_{\mathit{Mult}_i,\prec}$ has at least $2^{k/8}$ vertices. This shows that BDDs do not encode multiplication compactly irrespective of the variable ordering. ■

Exercise 3.16 {T3} [Deterministic finite automata and BDDs] Given a variable ordering, a boolean formula can be defined as a regular language over $\mathbb{B}$. A boolean expression p defines the region $[\![p]\!]$ that contains all states s that satisfy

p. Let $x_1 \prec \ldots \prec x_k$ be the enumeration of the variables according to $\prec$. Each state s is an assignment to the variables in X, and can be represented by the vector $s(x_1) \ldots s(x_k)$ over $\mathbb{B}$. Thus, $[\![p]\!]$ is a language over $\mathbb{B}$ that contains words of length k. Since $[\![p]\!]$ is a finite language, it is regular, and can be defined by a deterministic finite automaton (DFA). DFAs also have canonical forms: every regular language is accepted by a unique minimal DFA. This suggests that we can use DFAs as a representation of boolean functions. (1) Give an example of a boolean expression whose DFA representation is smaller than its BDD representation. (2) Give an example of a boolean expression whose BDD representation is smaller than its DFA representation. ■

3.2.3 Operations on BDDs

Let us turn our attention to implementing regions as BDDs. Every vertex of a BDD is itself a BDD rooted at that vertex. This suggests that a BDD can be represented by an index to a global data structure that stores vertices of all the BDDs such that no two vertices are isomorphic. There are two significant advantages to this scheme, as opposed to maintaining each BDD as an individual data structure. First, checking isomorphism, or equivalence, corresponds to comparing indices, and does not require traversal of the BDDs. Second, two non-isomorphic BDDS may have isomorphic subgraphs, and hence, can share vertices.

Let X be an ordered set of k propositions. The type of states is then $\mathbb{B}^k$. The type of BDDs is bdd, which is a pointer or an index to the global data structure *BddPool*. The type of *BddPool* is **set of bddnode**, and it stores the vertices of BDDs. The vertices of BDDs have type bddnode which equals $([1..k] \times \mathbf{bdd} \times \mathbf{bdd}) \cup \mathbb{B}$. The type bddnode supports the following operations:

Label: **bddnode** $\mapsto [1..k]$. The operation $Label(v)$, for an internal vertex v, returns the index of the variable labeling v.

Left: **bddnode** $\mapsto$ **bdd**. The operation $Left(v)$, for an internal vertex v, returns a pointer to the global data structure *BddPool* that points to the left-successor of v.

Right: **bddnode** $\mapsto$ **bdd**. The operation $Right(v)$, for an internal vertex v, returns a pointer to the global data structure *BddPool* that points to the right-successor of v.

The type **set of bddnode**, apart from usual operations such as Insert and IsMember, also supports

Index : **bddnode** $\mapsto$ **bdd**. For a vertex v in *BddPool*, $Index(v)$ returns a pointer to v.

function MakeVertex
Input: $i : [1..k]$, $B_0, B_1 :$ **bdd**.
Output: B : **bdd** such that $r(B)$ is equivalent to $(\neg x_i \wedge r(B_0)) \vee (x_i \wedge r(B_1))$.

begin
  if $B_0 = B_1$ then return B_0 fi;
  if $\neg \mathit{IsMember}((i, B_0, B_1), \mathit{BddPool})$ then
    $\mathit{Insert}((i, B_0, B_1), \mathit{BddPool})$ fi;
  return $\mathit{Index}((i, B_0, B_1))$
  end.

Figure 3.6: Creating BDD vertices

$[\cdot]$: **set of bddnode** $\times$ **bdd** $\mapsto$ **bddnode**. The operation $\mathit{BddPool}[B]$ returns the root vertex of the BDD B.

For such a representation, given a pointer B of type **bdd**, we write $r(B)$ to denote the propositional formula associated with the BDD that B points to. To avoid duplication of isomorphic nodes while manipulating BDDs, it is necessary that new vertices are created using the function MakeVertex of Figure 3.6. If no two vertices in the global set *BddPool* were isomorphic before an invocation of the function MakeVertex, then even after the invocation, no two vertices in *BddPool* are isomorphic. The global set *BddPool* initially contains only two terminal vertices, and internal vertices are added only using MakeVertex.

Exercise 3.17 {T3} [BDD with complement edges] A binary decision graph with complement edges (CBDG) is a binary decision graph B with an additional component that associates a boolean value with each right-edge. The predicate $r(v)$, for an internal vertex v, is redefined so that $r(v)$ equals $(\neg \mathit{label}(v) \wedge r(\mathit{left}(v))) \vee (\mathit{label}(v) \wedge r(\mathit{right}(v)))$ if the value associated with the right-edge of v is 1, and $(\neg \mathit{label}(v) \wedge r(\mathit{left}(v))) \vee (\mathit{label}(v) \wedge \neg r(\mathit{right}(v)))$ otherwise. Thus, when the right-edge is labeled with 0, we negate the function associated with the right-child. (1) Define binary decision digrams with complement edges (CBDD) as a subclass of CBDGs such that every boolean function has a unique representation as a CBDD. (2) Is there a function whose CBDD representation is smaller than its BDD representation? (3) Suppose we store vertices of all the functions in the same global pool. Show that CBDD representation uses less space than BDDs. (4) Show that the canonicity property is not possible if we allow complementing left-edges also. ■

To be able to build a BDD-representation of a given predicate, and to implement the primitives of the symbolic reachability algorithm, we need a way to construct conjunctions and disjunctions of BDDs. We give a recursive algorithm for obtaining conjunction of BDDs. The algorithm is shown in Figure 3.7.

Consider two vertices v and w, and we wish to compute the conjunction $r(v) \wedge r(w)$. If one of them is a terminal vertex, then the result can be determined immediately. For instance, if v is the terminal vertex labeled with false, then the conjunction is also false. If v is the terminal vertex labeled with true, then the conjunction is equivalent to $r(w)$.

The interesting case is when both v and w are internal vertices. Let i be the minimum of the indices labeling v and w. Then, x_i is the least variable in the support-set of $r(v) \wedge r(w)$. The label of the root of the conjunction is i, the left-successor is the BDD for $(r(v) \wedge r(w))[x_i := 0]$, and the right-successor is the BDD for $(r(v) \wedge r(w))[x_i := 1]$. Let us consider the left-successor. Observe the equivalence

$$(r(v) \wedge r(w))[x_i := 0] \equiv r(v)[x_i := 0] \wedge r(w)[x_i := 0] \quad (1).$$

If v is labeled with i, the BDD for $r(v)[x_i := 0]$ is the left-successor of v. If the label of v exceeds i, then the support-set of $r(v)$ does not contain x_i, and the BDD for $r(v)[x_i := 0]$ is v itself. The BDD for $r(w)[x_i := 0]$ is computed similarly, and then the function Conj is applied recursively to compute the conjunction (1).

The above described recursion may call the function Conj repeatedly with the same two arguments. To avoid unnecessary computation, a table is used that stores the arguments and the corresponding result of each invocation of Conj. When Conj is invoked with input arguments v and w, it first consults the table to check if the conjunction of $r(v)$ and $r(w)$ was previously computed. The actual recursive computation is performed only the first time, and the result is entered into the table.

A table data structure stores values that are indexed by keys. If the type of values stored is value, and the type of the indexing keys is key, then the type of the table is **table of key $\times$ value**. The abstract type table supports the retrieval and update operations like arrays: $T[i]$ is the value stored in the table T with the key i, and the assignment $T[i] := m$ updates the value stored in T for the key i. The constant table EmptyTable has the default value $\perp$ stored with every key. Tables can be implemented as arrays or as hash-tables. The table used by the algorithm uses a pair of BDDs as a key, and stores BDDs as values.

Let us analyze the time-complexity of Algorithm 3.2. Suppose the BDD pointed to by B_0 has m vertices and the BDD pointed to by B_1 has n vertices. Let us assume that the implementation of the set *BddPool* supports constant time membership tests and insertions, and the table Done supports constant-time

Algorithm 3.2 [Conjunction of BDDs]
Input: B_0, B_1 : **bdd**.
Output: B : **bdd** such that $r(B)$ is equivalent to $r(B_0) \wedge r(B_1)$.
Local: *Done* : **table of** (**bdd** $\times$ **bdd**) $\times$ **bdd**.

```
begin
  Done := EmptyTable;
  return Conj(B0, B1)
  end.

function Conj
input B0, B1 : bdd
output B : bdd
local v0, v1 : bddnode; B, B00, B01, B10, B11 : bdd; i, j : [1...k]

begin
  v0 := BddPool[B0];
  v1 := BddPool[B1];
  if v0 = 0 or v1 = 1 then return B0 fi;
  if v0 = 1 or v1 = 0 then return B1 fi;
  if Done[(B0, B1)] ≠⊥ then return Done[(B0, B1)] fi;
  if Done[(B1, B0)] ≠⊥ then return Done[(B1, B0)] fi;
  i := Label(v0); B00 := Left(v0); B01 := Right(v0);
  j := Label(v1); B10 := Left(v1); B11 := Right(v1);
  if i = j then B := MakeVertex(i,Conj(B00,B10), Conj(B01,B11))
   fi;
  if i < j then B := MakeVertex(j,Conj(B00,B1),Conj(B01,B1)) fi;
  if i > j then B := MakeVertex(j,Conj(B0,B10),Conj(B0,B11)) fi;
  Done[(B0, B1)] := B;
  return B
  end.
```

Figure 3.7: Conjunction of BDDs

creation, access, and update. Then, within each invocation of Conj, all the steps, apart from the recursive calls, are performed within constant time. Thus, the time-complexity of the algorithm is the same, within a constant factor, of the total number of invocations of Conj. For any pair of vertices, the function Conj produces two recursive calls only the first time Conj is invoked with this pair as input, and zero recursive calls during the subsequent invocations. This gives an overall time-complexity of $O(m \cdot n)$.

Proposition 3.5 [BDD conjunction] Given two BDDs B_0 and B_1, Algorithm 3.2 correctly computes the BDD for $r(B_0) \wedge r(B_1)$. If the BDD pointed to by B_0 has m vertices and the BDD pointed to by B_1 has n vertices, then the time-complexity of the algorithm is $O(m \cdot n)$.

Exercise 3.18 {T3} [Quadratic lower bound] The time complexity of Algorithm 3.2 is proportional to the product of the number of vertices in the component BDDs. Show that the size of the BDD representing conjunction of two BDDs grows as the product of the sizes of the components, in the worst case. ■

Exercise 3.19 {T3} [Cost of recomputation] Show that, removing the update step $Done[(B_0, B_1)] := B$ from Algorithm 3.2, makes the worst-case time complexity exponential. ■

Exercise 3.20 {T3} [From expressions to BDDs] Give an algorithm to construct the BDD-representation of a boolean expression given as a propositional formula. What is the time-complexity of the algorithm? ■

Exercise 3.21 {T3} [Substitution in BDDs] (1) Let p be a propositional formula, x be a variable, and $m \in \mathbb{B}$ be a value. Give an algorithm to construct the BDD representation of $p[x := m]$ from the BDD-representation of p. (2) Give algorithms for computing the disjunction and existential-quantifier elimination for BDDs. ■

Exercise 3.22 {T3} [BDD operations] The control schema underlying Algorithm 3.2 works for both conjunction and disjunction. What are the conditions on a binary operator on BDDs that make that control schema work? ■

3.2.4 Symbolic Search using BDDs

We have all the machinery to implement the symbolic search algorithm using BDDs as a representation for symbolic regions. It can be used immediately to solve the propositional invariant verification problem, and can be adopted to solve the enumerated invariant verification problem.

Exercise 3.23 {T3} [Symbolic verification of enumerated modules] An enumerated variable whose type contains k values can be encoded by $\lceil log\ k \rceil$ boolean variables. Give an algorithm which, given an enumerated invariant verification problem (P, p), constructs a propositional invariant verification problem (Q, q) such that (1) the answers to the two verification problems are identical, and (2) the description of the problem (Q, q) is at most $\lceil log\ k \rceil$ times the description of (P, p), where every variable of P has an enumerated type with at most k values. ■

We will consider some heuristics that are useful in different steps of applying Algorithm 3.1 for solving the invariant verification problem using BDDs.

Given the propositional invariant verification problem (P, p), the first step is to construct the symbolic representations for the target region p, the initial predicate q^I of P, and the transition predicate q^T of P. The BDD representations of a boolean expression can be exponentially larger, and is very sensitive to the ordering of variables. Heuristics are usually tailored to keep the representation of q^T small.

Computing with the frontier

Recall the symbolic algorithm of Figure 3.2 that searches the input graph in a breadth-first manner. A modified version of the algorithm is shown in Figure 3.8. In addition to the region σ^R containing the states known to be reachable, Algorithm 3.3 maintains an additional region σ^F called the frontier. In each iteration of the repeat loop, the frontier σ^F equals the subset of the reachable region σ^R containing only the newly discovered states. More precisely, after j iterations of the repeat loop, the region σ^R equals $post^{\leq j}(\sigma^I)$, and the frontier σ^F equals $post^{\leq j}(\sigma^I) \setminus post^{\leq j-1}(\sigma^I)$. Consequently, to find out which states are reachable in $j+1$ rounds, it suffices to compute the successor region of the frontier σ^F rather than the reachable region σ^R. In practice, Algorithm 3.8 typically outperforms Algorithm 3.2 in terms on computational resource requirements. The correctness statement for Algorithm 3.3 is identical to the one for Algorithm 3.1.

Theorem 3.2 [Symbolic graph search using the frontier] Let G be a transition graph, and let σ^T be a region of G. Algorithm 3.3, if it terminates, correctly solves the reachability problem (G, σ^T). Furthermore, if there exists $j \in \mathbb{N}$ such that (1) every reachable state is the sink of some initialized trajectory of length at most j (i.e. $\sigma^R = (\cup i \leq j.\, post^i(\sigma^I))$), or (2) some state in the target region σ^T is the sink of some initialized trajectory of length j (i.e. $\sigma^T \cap post^j(\sigma^I)$ is nonempty), then the algorithm terminates within j iterations of the repeat loop.

Remark 3.4 [Optimizing the frontier computation] For the correctness of Algorithm 3.3, it suffices if, after j iterations of the repeat loop, the frontier σ^A is any

Algorithm 3.3 [Symbolic Search using the Frontier]

> Input: a transition graph G, and a region σ^T of G.
> Output: the answer to the reachability problem (G, σ^T).

```
input G: symgraph;  σ^T: symreg;
local σ^R: symreg; σ^F: symreg;
begin
  σ^R := InitReg(G); σ^F := σ^R;
  repeat
    if σ^F ∩ σ^T ≠ EmptySet then return Yes fi;
    if PostReg(σ^F, G) ⊆ σ^R then return No fi;
    σ^R := σ^R ∪ PostReg(σ^F, G);
    σ^F := σ^R \ PostReg(σ^F, G)
    forever
  end.
```

Figure 3.8: Symbolic search using the frontier

region that contains at least $post^{\leq j}(\sigma^I) \setminus post^{\leq j-1}(\sigma^I)$ and at most $post^{\leq j}(\sigma^I)$. That is, the frontier should at least contain the newly discovered states, and should not contain any state not known to be reachable. This gives us freedom to choose the frontier so as to reduce the sise of its representation. ■

Choice of variable ordering

First, we need to choose an ordering $\prec$ of the variables in $X_P \cup X'_P$. One of the steps in the computation of the successor-region is to rename all the primed variables to unprimed variables. This renaming step can be implemented by renaming the labels of the internal vertices of the BDD if the ordering of the primed variables is consistent with the ordering of the corresponding unprimed variables. This gives us our first rule for choosing $\prec$;

> Variable Ordering Rule 1: For a reactive module P, choose the ordering $\prec$ of the variables $X_P \cup X'_P$ so that for all variables $x, y \in X_P$, $x \prec y$ iff $x' \prec y'$.

As the second rule of thumb to minimize the size of B_{q^T}, a variable should appear only after all the variables it depends on:

> Variable Ordering Rule 2: For a reactive module P, choose the ordering $\prec$ of the variables $X_P \cup X'_P$ so that (1) for every atom U in atoms_P, if $x \in \mathsf{read}X_U$ and $y \in \mathsf{ctr}X_U$ then $x \prec y'$, and (2) if the variable y awaits the variable x then $x' \prec y'$.

Since the set of atoms of a module is consistent, there exists an ordering that satisfies both the above rules.

Exercise 3.24 {T2} [Disjoint Dependence] Let p be a boolean function with support-set X, and let $\prec_1 = x_1, x_2, \ldots x_k$ be an optimal ordering of X for p. Let q be a boolean function with support-set Y, and let $\prec_2 = y_1, y_2 \ldots y_l$ be an optimal ordering of Y for q. Suppose $X_1 \cap X_2$ is empty.

(1) Show that the ordering $x_1, x_2, \ldots x_k, y_1, y_2 \ldots y_l$ is an optimal ordering for $p \vee q$ as well as for $p \wedge q$. (2) If the optimal BDD for p has m_1 vertices and the optimal BDD for q has m_2 vertices, how many vertices does the optimal BDD for $p \wedge q$ have? ■

Exercise 3.24 suggests that the variables that are related to each other should be clustered together. In particular, instead of ordering all the primed variables after all the unprimed variables, we can try to minimize the distance between a primed variable and the unprimed variables it depends on.

> Variable Ordering Rule 3: For a reactive module P, choose the ordering $\prec$ of the variables $X_P \cup X'_P$ so as to minimize the sum of the differences $j - i$ such that j-th variable according to $\prec$ is a primed variable x' that depends on the i-th variable according to $\prec$.

Exercise 3.25 {P2} [Ordering of module variables] The BDD for the transition relation of an atom is the disjunction of the BDDs for individual guarded assignments in its update command. Give a heuristic to order the variables that attempts to exploit this structure. Write an algorithm that, given a module P, constructs a variable ordering according to the heuristics discussed so far. ■

Partitioned transition relation

Another approach to constructing the BDD for the transition predicate is to avoid building it a priori.

Conjunctive Partitioning
A conjunctively-partitioned representation of a boolean expression p is a set $\{B_1, \ldots B_k\}$ of BDDs such that p is equivalent to the conjunction $r(B_1) \wedge \cdots \wedge r(B_k)$.

The total number of vertices in a conjunctively partitioned representation can be exponentially smaller than the number of vertices in the BDD for p itself. Since the transition predicate of a module is the conjunction of the update commands of its atoms, it leads to a natural conjunctively partitioned representation. This approach avoids building the BDD for the entire transition relation. Let us revisit the computation of the reachable region using symbolic search. Starting from the initial predicate $q_0 = q^I$, we successively compute the predicates q_i using

$$q_{i+1} \;=\; (\exists X.\, q_i \wedge q^T)[X' := X] \quad (2).$$

The computation (2) involves obtaining the conjunction $q_i \wedge q^T$. If q^T is conjunctively partitioned, $q_1^T \wedge \cdots \wedge q_k^T$, then we need to compute the conjunction $q_i \wedge q_1^T \wedge \cdots \wedge q_k^T$. Thus, it appears that we have only postponed the complexity of conjoining multiple BDDs, and in fact, we are now required to construct the conjunction at each step. The advantage is that, the size of $B_{q_i \wedge q^T}$ can be much smaller than the size of B_{q^T}. This is because q_i contains only reachable states, and thus constrains the source states for q^T. Thus, the conjunctively partitioned representation is an on-the-fly symbolic representation. While computing the BDD for the conjunction $q_i \wedge q_1^T \wedge \cdots q_k^T$, we do not need to construct B_{q^T} first. We can compute the conjunction from left to right, starting with the construction of $B_{q_i \wedge q_1^T}$.

Remark 3.5 [Two-level representations] The idea of on-the-fly representation of the transition relation can be extended further. Each conjunct q_i^T, representing the update command of a single atom, is a disjunction of predicates obtained from individual guarded assignments. When the number of guarded assignments in a single update command is large, it may not be suitable to construct the BDD for the update command, and maintain it in a disjunctively-partitioned form. It is possible to maintain two-level, or even multi-level, BDD-representation of the transition predicate, and manipulate it only during the computation of (2). ■

Early quantifier elimination

Observe the equivalence

$$\exists x.\, p \wedge q \;\equiv\; p \wedge \exists x.\, q \qquad \text{if } x \text{ is not in the support-set of } p$$

If x is a support variable for q, then $B_{\exists x.\, q}$ can have less number of vertices than B_q. This implies that to compute $B_{\exists x.\, p \wedge q}$ from B_p and B_q, if x is not a support variable of p, the best strategy is to first compute $B_{\exists x.\, q}$ and then conjoin it with B_p. This strategy to apply the projection operation before conjunction is called early quantifier-elimination.

The computation (2) requires the computation of the projection of a conjunction of BDDs onto a set of variables, and thus, demands the use of early quantifier-elimination. It is even more effective if we are computing the reachable region of the reduced transition graph of a module P. Then the transition predicate q^T is itself a projection of the transition predicate of G_P onto the latched variables.

The region predicate q_i is an expression over the latched variables $latchX_P$, and (2) is rewritten as

$$q_{i+1} \;=\; (\exists X_P.\, \exists X'_P \backslash latchX'_P.\, q_i \,\wedge\, q_1^T \,\wedge\, \cdots \,\wedge\, q_k^T)[latchX'_P := latchX_P] \quad (3).$$

Example 3.6 [Early-quantifier elimination in computing PostReg] Consider a synchronous 3-bit counter that is incremented in every round. The variables of the module are out_0, out_1, out_2. The update of the bit out_0 is specified by

$$q_0^T:\; out'_0 \leftrightarrow \neg out_0,$$

of the bit out_1 by

$$q_1^T:\; out'_1 \leftrightarrow out_0 \oplus out_1$$

and of the bit out_2 by

$$q_2^T:\; out'_2 \leftrightarrow (out_0 \,\wedge\, out_1) \oplus out_2,$$

where $\oplus$ denotes exclusive-or operation. The transition predicate is the conjunction $q_0^T \wedge q_1^T \wedge q_2^T$. A good variable-ordering according to our rules is

$$out_0 \prec out'_0 \prec out_1 \prec out'_1 \prec out_2 \prec out'_2.$$

We choose not to construct q^T a priori, but to maintain it in a conjunctively partitioned form. Consider a predicate p, and we wish to compute the predicate

$$\exists\{out_0, out_1, out_2\}.\, (p \,\wedge\, q_0^T \,\wedge\, q_1^T \,\wedge\, q_2^T). \quad (1)$$

If we first compute the conjunction $p \wedge q_0^T$, we cannot eliminate any of the variables. However, since conjunction is associative and commutative, we can choose the ordering of the conjuncts. In particular, if we first conjunct p and q_2^T, then none of the remaining conjuncts depend on out_2, and hence, we can eliminate out_2. Thus, (1) can be rewritten to

$$\exists\, out_0.\, (q_0^T \,\wedge\, \exists\, out_1.\, (q_1^T \,\wedge\, \exists\, out_2.\, (q_2^T \,\wedge\, p))).$$

Thus, the support-sets of various BDDs are examined to determine an ordering of the conjuncts so as to eliminate as many variables as early as possible. ■

Exercise 3.26 {T3} [Don't care simplification] This exercise describes an effective heuristic for simplifying each conjunct of the transition relation with respect to the current reachable region Given predicates p and q over the set X of variables, the predicate r over X is said to be a p-simplification of q if $p \rightarrow (q \leftrightarrow r)$ is valid. Thus, a p-simplification of q must include states that satisfy both p and q, must exclude states that satisfy p but not q, and can treat the remaining states as "don't care" states. Observe that a p-simplification of q is not unique, and its BDD representation can have much smaller size compared to the BDD representation of q.

(1) Give an algorithm that computes, given the BDD representations of two predicates p and q, BDD representation of some p-simplification of q. The objective should be to reduce the size of the output BDD by exploiting the freedom afforded by "don't care" when p is false. (2) Show that the conjunction $q \wedge q_1^T \wedge \cdots q_k^T$ during the computation of the successor region can be replaced by $q \wedge r_1 \wedge \cdots r_k$ where each r_j is a q-simplification of the conjunct q_j^T. Observe that this strategy simplifies different conjuncts independently of each other, rather than sequentially as in early quantifier elimination, and hence, is not sensitive to the ordering of the conjuncts. ■

Dynamic variable reordering

As Algorithm 3.1 computes the reachable region using successive approximations, the BDD representing q_i grows with i, and successive applications of *PostReg* require more and more time. If the number of vertices exceeds beyond a threshhold, we can attempt to reduce its size by choosing a new ordering of the variables. This step is called dynamic variable reordering.

If we want to switch the ordering of two adjacent variables, its effect on the BDD is local: if we switch the variables x_i and x_{i+1}, then the structure of the BDD for vertices labeled less than i or greater than $i+1$ does not change.

Exercise 3.27 {T4} [Swapping of Variables] Consider a boolean function p, and its $B_{p,\prec}$ using the linear order $x_1 \prec \cdots \prec x_n$. For $1 \leq i < n$, let $\prec_{[i/i+1]}$ be the linear order

$$x_1 \prec_{[i/i+1]} \cdots \prec_{[i/i+1]} x_{i+1} \prec_{[i/i+1]} x_i \prec_{[i/i+1]} x_{i+1} \prec_{[i/i+1]} \cdots x_n$$

obtained by swapping the order of x_i and x_{i+1}. Show that (1) a vertex of $B_{p,\prec}$ that is labeled with an index j that is greater than $i+1$ is also a vertex in $B_{p,\prec_{[i/i+1]}}$, and (2) the subgraph of $B_{p,\prec}$ containing vertices labeled with indices less than i is isomorphic to the subgraph of $B_{p,\prec_{[i/i+1]}}$ containing vertices labeled with indices less than i. Give an algorithm to construct $B_{p,\prec_{[i/i+1]}}$. What is the complexity of your algorithm? ■

This suggests a variety of greedy heuristics for reordering. Suppose i is the index such that maximum number of vertices of B are labeled with i. Then, we can try swapping x_i with x_{i+1} or x_i with x_{i-1}. If one of the swaps reduces the size of the BDD, we choose the resulting order. Alternatively, one can try successive local swaps till BDD size is reduced. Note that if we update the ordering of the variables for one BDD, all the other BDDs need to be updated. Thus, dynamic variable reordering is a costly step, and is invoked only in extreme cases. Efficient memory management techniques for garbage collection of BDD-vertices not in use is also essential in practice.

Appendix: Notation

Orders and fixpoints

A binary relation $\preceq$ over a set A is a preorder if it is reflexive and transitive, and a partial order if it is reflexive, transitive, and antisymmetric. Let $\preceq$ be a preorder on A, and let $\succeq$ be the inverse of $\preceq$. If $\preceq$ is a partial order, then so is $\succeq$; if $\preceq$ is an equivalence, then $\succeq = \preceq$. The preorder $\preceq$ induces the equivalence $\preceq \cap \succeq$, which is called the kernel of $\preceq$. Given $B \subseteq A$ and $a \in A$, we write $B \preceq a$ if for all $b \in B$, $b \preceq a$; in this case, a is an upper $\preceq$-bound for B. An upper $\succeq$-bound for B is called a lower $\preceq$-bound for B. Moreover, a is a least upper $\preceq$-bound for B if (1) a is an upper $\preceq$-bound for B, and (2) a is a lower $\preceq$-bound for the set of upper $\preceq$-bounds for B. A least upper $\succeq$-bound is called a greatest lower $\preceq$-bound. If $\preceq$ is a partial order, then all least upper $\preceq$-bounds and all greatest lower $\preceq$-bounds are unique. The partial order $\preceq$ is complete if all subsets of A have least upper $\preceq$-bounds (and hence greatest lower $\preceq$-bounds); in this case we write $\bigvee B$ for the least upper $\preceq$-bound for B, and $\bigwedge B$ for the greatest lower $\preceq$-bound for B. Every complete partial order $\preceq$ has the unique lower $\preceq$-bound $\bigvee \emptyset$ for A, called bottom, and the unique upper $\preceq$-bound $\bigwedge \emptyset$ for A, called top. For example, the subset relation $\subseteq$ is a complete partial order on the powerset $\mathbf{2}^C$ of any set C. In this case, the least upper $\subseteq$-bound for a set E of subsets of C is the union $\bigcup E$; the greatest lower $\subseteq$-bound for E is the intersection $\bigcap E$; the bottom is the empty set $\emptyset$; and the top is the entire set C.

Let $\preceq$ be a complete partial order on A with the bottom $\perp$ and the top $\top$, and let f be a function from A to A. The function f is monotonic if for all $a, b \in A$, if $a \preceq b$ then $f(a) \preceq f(b)$. The argument $a \in A$ is a fixpoint of f if $f(a) = a$. If f is monotonic, then $\preceq$ is a complete partial order on the fixpoints of f.[1] The bottom fixpoint, denoted μf, is called the least fixpoint of f; the top fixpoint, denoted νf, is the greatest fixpoint of f. The function f is $\bigvee$-continuous if for all $B \subseteq A$, $f(\bigvee B) = \bigvee f(B)$, and $\bigwedge$-continuous if for all $B \subseteq A$, $f(\bigwedge B) = \bigwedge f(B)$. If f is monotonic and $\bigvee$-continuous, then $\mu f = \bigvee\{f^\kappa(\perp) \mid \kappa \in \mathbb{O}\}$.[2 3] If, in addition, A is countable, then $\mu f = \bigvee\{f^i(\perp) \mid i \in \mathbb{N}\}$; if A is finite, then there is a natural number i such that $\mu f = f^i(\perp)$. Analogous results apply to the greatest fixpoint of a monotonic and $\bigwedge$-continuous function.

Exercise 3.28 {} [Fixpoint theorems] Prove all claims made in the previous paragraph. ■

[1] This is the Knaster-Tarski fixpoint theorem.
[2] By $\mathbb{O}$, we denote the set of ordinals. For a limit ordinal λ, let $f^\lambda(a) = \bigvee\{f^\kappa(a) \mid \kappa < \lambda\}$.
[3] This is the Kleene fixpoint theorem.