Formal Verification

Lecture 8: Operations on Binary Decision Diagrams (BDDs)

Jacques Fleuriot
jdf@inf..ac.uk

Diagrams from Huth & Ryan, LiCS, 2nd Ed.
Recap

- Previously:
  - (Reduced, Ordered) Binary Decision Diagrams ((RO)BDDs)

- This time:
  - Operations on ROBDDs
    - reduce, apply, restrict, exists
  - Symbolic Model Checking with BDDs
**Binary Decision Diagrams**

Binary Decision Diagrams: DAGs, such that
- Unique root node
- Variables on non-terminal nodes
- Truth-values on terminal nodes
- Exactly two edges from each non-terminal node, labelled 0, 1

Some notation, for a given BDD node $n$:
- If $n$ is a non-terminal node:
  - $\text{var}(n)$ — the variable label on node $n$;
  - $\text{lo}(n)$ — the node reached by following the 0 edge from $n$;
  - $\text{hi}(n)$ — the node reached by following the 1 edge from $n$;
- If $n$ is a terminal node:
  - $\text{val}(n)$ — the truth value labelling $n$

For a BDD $B$, the root node is called $\text{root}(B)$. 
reduce constructs a ROBDD from an OBDD.

1. Label each OBDD node $n$ with an integer $\text{id}(n)$,
2. in a single bottom-up pass, such that:
3. two OBDD nodes $m$ and $n$ have the same label ($\text{id}(m) = \text{id}(n)$) if and only if $m$ and $n$ represent the same boolean function.

The ROBDD is then created by using one node from each class of nodes with the same label.
Assignment of labels follows the rules for performing reductions.

To label a node $n$:

- **Remove duplicate terminals:**
  if $n$ is a terminal node (i.e., 0 or 1), then set $\text{id}(n)$ to be $\text{val}(n)$.

- **Remove redundant tests:**
  if $\text{id(lo}(n)) = \text{id(hi}(n))$ then set $\text{id}(n)$ to be $\text{id(lo}(n))$.

- **Remove duplicate nodes:**
  if there exists a node $m$ that has already been labelled such that
  \[
  \begin{cases}
  \text{var}(m) = \text{var}(n) \\
  \text{lo}(m) = \text{lo}(n) \\
  \text{hi}(m) = \text{hi}(n)
  \end{cases}
  \]
  set $\text{id}(n)$ to $\text{id}(m)$.

  Use a hashtable with $\langle \text{var}(n), \text{lo}(n), \text{hi}(n) \rangle$ keys for $O(1)$ lookup time.

- Otherwise, set $\text{id}(n)$ to an unused number.
In practice, labelling and construction are interleaved.
In practice, labelling and construction are interleaved.
In practice, labelling and construction are interleaved.
reduce Example

In practice, labelling and construction are interleaved.
In practice, labelling and construction are interleaved.
In practice, labelling and construction are interleaved.
In practice, labelling and construction are interleaved.
In practice, labelling and construction are interleaved.
In practice, labelling and construction are interleaved.
reduce Example

Reduces to

In practice, labelling and construction are interleaved.
reduce Example

In practice, labelling and construction are interleaved.
Given compatible OBDDs $B_f$ and $B_g$ that represent formulas $f$ and $g$, apply($\square$, $B_f$, $B_g$) computes an OBDD representing $f \square g$.

- where $\square$ represents some binary operation on boolean formulas
  *for example, $\land, \lor, \oplus$*
- Unary operations can be handled too.
  *for example, negation: $\neg x = x \oplus 1$*
apply: Shannon expansions

For any boolean formula $f$ and variable $x$, it can be written as:

$$f \equiv (\neg x \land f[0/x]) \lor (x \land f[1/x])$$

This is the **Shannon expansion** of $f$ (originally due to G. Boole).
**apply: Shannon expansions**

For any boolean formula $f$ and variable $x$, it can be written as:

$$f \equiv (\neg x \land f[0/x]) \lor (x \land f[1/x])$$

This is the **Shannon expansion** of $f$ (originally due to G. Boole).

In particular: $f \square g$ can be expanded like so:

$$f \square g \equiv (\neg x \land (f[0/x] \square g[0/x])) \lor (x \land (f[1/x] \square g[1/x]))$$
apply: Shannon expansions

For any boolean formula $f$ and variable $x$, it can be written as:

$$f \equiv (\neg x \land f[0/x]) \lor (x \land f[1/x])$$

This is the Shannon expansion of $f$ (originally due to G. Boole).

In particular: $f \Box g$ can be expanded like so:

$$f \Box g \equiv (\neg x \land (f[0/x] \Box g[0/x])) \lor (x \land (f[1/x] \Box g[1/x]))$$

If a BDD $\node x$ represents a boolean function $f$, then:

1. $B$ represents $f[0/x]$ and $B'$ represents $f[1/x]$; and
2. The BDD is effectively a compressed representation of $f$ in Shannon normal form.

So: implement apply recursively on the structure of the BDDs.
apply: cases

apply(□, \( x \), \( x \)) = apply(□, B, C) \cdot apply(□, B', C')

apply(□, \( x \), C) = apply(□, B, C) \cdot apply(□, B', C')
when \( C \) is terminal node, or non-terminal with \( \text{var(root}(C)) > x \)

apply(□, B, \( x \)) = apply(□, B, C) \cdot apply(□, B, C')
when \( B \) is terminal node, or non-terminal with \( \text{var(root}(B)) > x \)

apply(□, u, v) = u \cdot v
apply: example

Compute apply(∨, B_f, B_g), where B_f and B_g are:
apply: recursive calls
apply: memoisation

The recursive apply implementation will generate an OBBD.
  ▶ Apply reduce to convert it back to a ROBDD.

However, as can be seen from the tree of recursive calls, there are many calls to apply with the same arguments.
  ▶ Each invocation of apply where at least one of the arguments is non-terminal generates two further calls to apply: the number of calls is worst-case exponential in the sizes of the original diagrams.

We are not taking into account the sharing in BDDs.

We can greatly improve the run-time by using memoisation: remembering the results of previous calls.
apply: memoised recursive calls

Memoisation results in at most $|B_f| \cdot |B_g|$ calls to apply.
apply: Result

If we are careful to never create the same BDD node twice (using the same lookup table technique as reduce), then with memoisation, we automatically get a reduced BDD:
Other Operations

\texttt{restrict}(0, x, B_f) \text{ computes ROBDD for } f[0/x]

1. For each node \( n \) labelled with \( x \), incoming edges are redirected to \( lo(n) \), and the node \( n \) is removed.
2. Resulting BDD then reduced with \texttt{reduce}.
3. (again, \texttt{reduce} can be interleaved with the removal.)

\texttt{exists}(x, B_f) \text{ computes ROBDD for } \exists x. \ f.

1. Uses the identity

\[(\exists x. \ f) \equiv f[0/x] \lor f[1/x]\]

2. Realised using the \texttt{restrict} and \texttt{apply} functions:

\[\text{apply} (\lor, \text{restrict}(0, x, B_f), \text{restrict}(1, x, B_f))\]
# Time Complexities

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Input OBDDs</th>
<th>Output OBDD</th>
<th>Time complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>reduce</td>
<td>$B$</td>
<td>reduced $B$</td>
<td>$O(</td>
</tr>
<tr>
<td>apply</td>
<td>$B_f, B_g$ (reduced)</td>
<td>$B_f \boxtimes g$ (reduced)</td>
<td>$O(</td>
</tr>
<tr>
<td>restrict</td>
<td>$B_f$ (reduced)</td>
<td>$B_f[0/x]$ or $B_f[1/x]$ (red’d)</td>
<td>$O(</td>
</tr>
<tr>
<td>$\exists$</td>
<td>$B_f$ (reduced)</td>
<td>$B_{\exists x_1 \ldots x_n} f$ (reduced)</td>
<td>NP-complete</td>
</tr>
</tbody>
</table>

H&R, Figure 6.23
Implementing CTL Model Checking using BDDs

Recall:

1. CTL model checking computes a set of states $[\phi]$ for every sub-formula $\phi$ of the original formula.
2. Sets of states will be represented using ROBDDs

States are represented by boolean vectors $\langle v_1, \ldots, v_n \rangle$ (i.e. $v_i \in \{0, 1\}$).

Sets of states are represented using ROBDDs on $n$ variables $x_1, \ldots, x_n$ (the Atoms) that describe the characteristic function of the set (see H&R 6.3.1 for a detailed description).

- Set operations ($\cap$, $\cup$, $\neg$) implemented using the operations on BBDs

For example, the definition

$$[[\phi \land \psi]] = [[\phi]] \cap [[\psi]]$$

is implemented by:

$$B[[\phi \land \psi]] = \text{apply}(\land, B[[\phi]], B[[\psi]])$$
Implementing CTL Model Checking using BDDs

Transition relations \( (\rightarrow) \subseteq S \times S \) are represented by ROBDDs on \( 2n \) variables.

- If the variables \( x_1, \ldots, x_n \) describe the current state, and the variables \( x'_1, \ldots, x'_n \) describe the next state, then a good ordering is \( x_1, x'_1, x_2, x'_2, \ldots, x_n, x'_n \) (interleaving).

When translating from the model description, the boolean formulas describing the:

1. initial state set
2. transition relation
3. defined variables

are translated into ROBDDs by using the \texttt{apply} algorithm, following the structure of the original formula.

This avoids exponential blow-up from first constructing a decision tree and then reducing.
Implementing CTL Model Checking using BDDs

The function applications

\[
\begin{align*}
\text{pre}_\exists(Y) & = \{ s \in S \mid \exists s' \in S. (s \rightarrow s') \land s' \in Y \} \\
\text{pre}_\forall(Y) & = \{ s \in S \mid \forall s' \in S. (s \rightarrow s') \rightarrow s' \in Y \}
\end{align*}
\]

are implemented using BDDs like so:

\[
B_{\text{pre}_\exists(Y)} = \text{exists}(\vec{x'}, \text{apply}(\land, B_{\rightarrow}, B_{Y'}))
\]

where

- \( B_{\rightarrow} \) is the ROBDD representing the transition relation \( \rightarrow \);
- \( B_{Y'} \) is the ROBDD representing the set \( Y \) with the variables \( x_1, \ldots, x_n \) renamed to \( x'_1, \ldots, x'_n \).

And:

\[
\text{pre}_\forall(Y) = S - \text{pre}_\exists(S - Y)
\]

where \( S - Y \) is implemented by negation (via apply).
Implementing CTL Model Checking using BDDs

To implement the temporal connectives, we compute fix points.

\[
\begin{align*}
[EF \phi] &= \mu Y. [\phi] \cup \text{pre}_\exists(Y) \\
[EG \phi] &= \nu Y. [\phi] \cap \text{pre}_\exists(Y)
\end{align*}
\]

By Knaster-Tarski, we know that:

- \( F|S| (\emptyset) \) is the least fixed point of \( F: \mu Y. F(Y) \)
- \( F|S| (S) \) is the greatest fixed point of \( F: \nu Y. F(Y) \)

Compute \([EF \phi]\) using the sequence (of ROBDDs)

\[
Y^0 = \emptyset, \quad Y^1 = [\phi] \cup \text{pre}_\exists(\emptyset), \quad Y^2 = [\phi] \cup \text{pre}_\exists([\phi] \cup \text{pre}_\exists(\emptyset)), \ldots
\]

Usually, we won’t need \(|S|\) steps: we can stop when \( Y_i = Y_{i+1} \)

- This check is very cheap with ROBDDs.
Summary

- Operations on BDDs (H&R 6.2)
  - reduce
  - apply
  - restrict, exists

- Symbolic Model Checking (H&R 6.3)
  - Representing states and transitions as BDDs
  - Implementing the CTL MC algorithm with BDDs