Agenda

1. Graph Theory: Some Definitions
2. Introduction to Petri Nets
3. Introduction to Computation Tree Logic and related model checking techniques
4. Introduction to Binary Decision Diagrams
5. Timed Automata and timed CTL

Part I

Binary Decision Diagrams
Binäre Entscheidungsdiagramme sind bi-partite, ungewichtete, zyklenfreie Digraphen, in denen ein jeder inneren Knoten jeweils genau 2 Nachfolger hat, nahezu den 0-Nachfolger und den 1-Nachfolger.


In den letzten 2 Dekaden sind BDDs sehr gründlich erforscht worden und es existieren viele abgeleitete Formen, sowie effiziente Algorithmen zu ihrer Manipulierung.

Letztendlich bilden BDDs und ihre verwandten Datentypen ein wichtiges Fundament im Very-large-scale integration (VLSI) Design und im Bereich des Model checkings.

Da BDDs letztendlich eine Implementierung einer endlichen Booleschen Algebra darstellen spricht man in diesem Zusammenhang oft auch von symbolischen Verfahren, bspw. vom symbolischen Model checking.

A Binary Decision Tree (BDT) is a bi-partite tree consisting of a set of inner nodes ($\mathcal{N}_{NT}$) and a set of terminal nodes ($\mathcal{N}_T$) with $\mathcal{N} := \mathcal{N}_{NT} \cup \mathcal{N}_T$.

- Nodes are connected via 1- and 0-edges: $\rightarrow \subseteq \mathcal{N}_{NT} \times \mathcal{N}$ and $\leftarrow \subseteq \mathcal{N}_{NT} \times \mathcal{N}$
- We read the tree from top to bottom, hence we can omit the arrow heads
- Each inner node (circle) is associated with a node label $n_i$ and a variable $x_i$, e.g. $\text{var}(n_6) = x_3$
- A dashed line leads to the 0-successor, the solid line to the 1-successor, e.g. $\text{child}_0(n_1) = n_3$; $\text{child}_1(n_1) = n_2$
- Each terminal node is associated with a function value from $\mathbb{B} := \{0, 1\}$, e.g. $\text{value}(t_1) = 0$

For algorithmically working with BDDs it turns out that they should be ordered w. r. t. the variables of $\mathcal{V}$.

To do so one simply defines a total order $\preceq \subseteq \mathcal{V} \times \mathcal{V}$ and requires

$$\forall n \in \mathcal{N}_{NT} : n = \text{child}_{0,1}(m) \Rightarrow \text{var}(m) \prec \text{var}(n)$$

What is the Boolean function represented by the BDT?

What is the space complexity for representing Boolean functions with BDT?

Shannon expansion for Boolean functions:

$$f(x_1, \ldots, x_n) = x_1 \cdot f_1(x_2, \ldots, x_n) + (1 - x_1) \cdot f_0(x_2, \ldots, x_n)$$

instead of the Boolean operators ($\neg, \lor, \land$) we employ their arithmetic counterparts, e.g. $\neg x_1 \equiv (1 - x_1)$, etc.

The recursion tree of a Shannon expansion is exactly what is represented by a BDT. Let BDT-node $k$ be labelled with variable $x_1$. According to the Shannon expansion it represents the $n$-ary Boolean function $f(x_1, \ldots, x_n)$.

- Its 1-successor represents than $f_1(x_2, \ldots, x_n)$ and its 0-successor represents function $f_0(x_2, \ldots, x_n)$.
- Function $f_0(x_1, \ldots, 1, x_i, \ldots, x_n)$ is denoted 1-cofactor of function $f$ w. r. t. variable $x_i$.
- Function $f_1(x_1, \ldots, 0, x_i, \ldots, x_n)$ is denoted 0-cofactor of function $f$ w. r. t. variable $x_i$.
- For the co-factors we also adapt the notation $f\mid_{x_i} = b$ with $b \in \{0, 1\}$
- A terminal node represents the 0-ary, constant 0 or 1-function.
According to the above discussion each BDT-node represents a Boolean function.

Let node $n$ represent function $f_n$ and let node $k$ represent function $f_k$:

Question 1.1: How can we decide if $f_n \equiv f_k$ holds?

We (recursively) define the equivalence relation $\equiv$ on the set of BDT-nodes ($N = N_{NT} \cup N_T$) as follows:

- for two terminal BDT-nodes $t, p \in N_T$:
  \[ t \equiv p \iff \text{value}(t) = \text{value}(p) \]
- for two non-terminal BDT-nodes $n, k \in N_{NT}$:
  \[ n \equiv k \iff \text{child}_0(n) \equiv \text{child}_0(k) \land \text{child}_1(n) \equiv \text{child}_1(k) \]

equivalent, i.e., $p \equiv t$ iff According to the above discussion each BDT-node represents a Boolean function.

Question 1.2: How does this effect the size of the obtained graphs?
Specification models and their analysis: Binary Decision Diagramms 13–36

1x1n
2n3n
6n 5n
2x
3x
0
7n
1

Specification models and their analysis: Binary Decision Diagramms 14–36

1x1n
2n3n
6n 5n
2x
3x
0
7n
1

Specification models and their analysis: Binary Decision Diagramms 15–36

1x1n
2n3n
6n 5n
2x
3x
0
7n
1

Specification models and their analysis: Binary Decision Diagramms 16–36

1x1n
2n3n
6n 5n
2x
3x
0
7n
1

Binary Decision Tree: Isomorphism of nodes

Binary Decision Tree: Isomorphism of nodes

Binary Decision Tree: Isomorphism of nodes

Binary Decision Tree: Isomorphism of nodes
Specification models and their analysis: Binary Decision Diagrams 17–36

Specification models and their analysis: Binary Decision Diagrams 18–36
Binary Decision Tree: Isomorphism of nodes

The reduction is applied on the fly, i.e., each allocated node is unique. Hence application of an a posteriori reduction not necessary.

As we will see later, uniqueness of nodes is not only a key to memory efficiency but also to run-time efficiency w.r.t. the manipulation of BDDs.

Specification models and their analysis: Binary Decision Diagrams 21–36

BDD: Multi-rooted BDDs

Uniqueness of BDD nodes allows one to share sub-graphs among different BDDs yielding multi-rooted BDDs:

The reduction is applied on the fly, i.e., each allocated node is unique.

Hence application of an a posteriori reduction not necessary.

As we will see later, uniqueness of nodes is not only a key to memory efficiency but also to run-time efficiency w.r.t. the manipulation of BDDs.

Specification models and their analysis: Binary Decision Diagrams 21–36

Binary Decision Tree: Isomorphism of nodes

A node \( n \in N_{NT} \) is denoted don’t-care (dnc) node iff

\[
\text{child}_0(n) = \text{child}_1(n)
\]

As shown by the example the Shannon-expansion yields, that such nodes can safely be omitted once one allocates BDDs.

Specification models and their analysis: Binary Decision Diagrams 22–36

--- Question 1.3: Can we do more, e.g. apply Shannon for function \( f_3 \)?

Specification models and their analysis: Binary Decision Diagrams 23–36

Binary Decision Diagram: Don’t care nodes

A node \( n \in N_{NT} \) is denoted don’t-care (dnc) node iff

\[
\text{child}_0(n) = \text{child}_1(n)
\]

As shown by the example the Shannon-expansion yields, that such nodes can safely be omitted once one allocates BDDs.
A node $n \in N_{NT}$ is denoted don’t-care (dnc) node iff

$\text{child}_0(n) = \text{child}_1(n)$ holds

As shown by the example the Shannon-expansion yields, that such nodes can safely be omitted once one allocates BDDs.

Specification models and their analysis: Binary Decision Diagrams 25–36
A reduced ordered Binary Decision Diagram \( B < V, \prec \) is a 5-tuple \( \{N, \text{value}, \text{var}, \text{child}_0, \text{child}_1\} \) where

- \( V \) is a finite and non-empty set of boolean variables with the fixed ordering relation \( \prec \subseteq V \times V \) defined one.
- \( N = N_T \cup N_{NT} \) is a finite non-empty set of nodes, consisting of the set of terminal nodes \( N_T \) and non-terminal nodes \( N_{NT} \), with \( N_T \cap N_{NT} = \emptyset \).

The following functions are defined:

- the value-returning function \( \text{value} : N_T \mapsto B \) for each terminal node,
- the variable-returning function \( \text{var} : N_{NT} \mapsto V \) for each non-terminal node,
- the child node-returning functions \( \text{child}_0, \text{child}_1 : N_{NT} \mapsto N \) for each non-terminal node, and
- the root node-returning function \( \text{getRoot} : B \mapsto N \).

For the BDD to be ordered the following constraint must hold:

\[
\forall u \in N_{NT} : \ 
\text{child}_1(u) \in N_{NT} : \text{var}(\text{child}_1(u)) \succ \text{var}(u) \\
\text{child}_0(u) \in N_{NT} : \text{var}(\text{child}_0(u)) \succ \text{var}(u).
\]

A BDD is denoted reduced iff the following conditions apply:

(a) Isomorphism rule: No isomorphic nodes; i.e.

(i) Non-terminal case: \( \forall n, m \in N_{NT} : n \neq m \Rightarrow (\text{var}(n) \neq \text{var}(m)) \lor (\text{child}_1(n) \neq \text{child}_1(m) \lor \text{child}_0(n) \neq \text{child}_0(m)) \)

(ii) Terminal case: \( \forall n, m \in N_T : n \neq m \Rightarrow (\text{value}(n) \neq \text{value}(m)) \)

(b) Dnc-rule: No don’t care nodes: \( \exists u \in N_{NT} : \text{child}_0(u) = \text{child}_1(u) \).

Reduced ordered BDDs are (strongly) canonical representations for Boolean Functions, thus each Boolean function \( f \) produces its own BDD \( B_f \).

\[
f \neq g \iff B_f \neq B_g
\]

\[\rightarrow\text{ Question 1.4: Why can equivalenz testing be done in constant time?}\]

Consider the following two Boolean functions:

\[
f : = \neg d a b + \neg a d \neg c + a b d + \neg a \neg c \neg d \\
g : = \neg a \neg c b + c b a + \neg b \neg a \neg c + a \neg b c
\]

\[\rightarrow\text{ Question 1.5: Are } f \text{ and } g \text{ equivalent? Please justify by making use of BDDs}\]

\[\rightarrow\text{ Excursion 1.1: Proof of canonicity (on the black board)}\]

For making use of BDDs in an algebraic framework it is neccessary to being capable of efficiently applying operators to them, s.t. the obtained BDD represents the resp. function. Hence any \( n \)-ary operator applicable to \( n \) Boolean functions should be applicable to their \( n \) BDD-based representations.

In the following we consider 1-ary and 2-ary (binary) operators, s.t.

\[
\neg f \quad \leadsto \quad \text{Negate}(B_f) \\
f + g \quad \leadsto \quad \text{Plus}(B_f, B_g) \\
f \cdot g \quad \leadsto \quad \text{Mult}(B_f, B_g)
\]
BDD-based algorithms: Negation

\[ \text{Negate}(\text{node } n) \]

(0) IF \( n \in N_T \) THEN
   RETURN(makeTerminal(1 - value(n)))
(1) ELSE
   (2) node \( t := \text{Negate}(\text{child}_1(n)) \)
   (3) node \( e := \text{Negate}(\text{child}_0(n)) \)
   (4) IF \( t = e \) THEN RETURN(t)
   (5) ELSE RETURN(makeNode(var(n), t, e))
END

--- Question 1.6: Construct the recursion tree for Negate and the BDD depicted above

--- Example 1.1:
Consider \( f_1 := \neg x_1 x_2 \) and \( f_2 := \neg x_1 x_3 \). Please give the BDDs for \( f_1 \) and \( f_2 \), construct the recursion tree for \( f_1 \land f_2 \) and give the resulting BDD.

--- Example 1.2: BDDs

In total the so far discussed techniques gives us a framework for efficiently representing and manipulating Boolean functions. This is the basis for representing and verifying systems such as

- Symbolic analysis of switching functions
- Symbolic reachability set generation, especially in case of Petri nets
- Symbolic CTL model checking