Specification models and their analysis

Kai Lampka

November 19, 2009

Institut für Technische Informatik und Kommunikationsnetze

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 ◁

Part I

Binary Decision Diagramms

Agenda

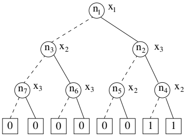
1. Graph Theoretic Foundations
2. Petri Nets
3. Computation Tree Logic and related model checking techniques
4. Binary Decision Diagrams ◁
5. Timed Automata and timed CTL

Binary Decision Diagrams: Zusammenfassung

- Binäre Entscheidungsdiagramme sind bi-partite, ungewichtete, zyklenfreie Digraphen, in denen ein jeder inneren Knoten jeweils genau 2 Nachfolger hat, naemlich den 0-Nachfolger und den 1-Nachfolger.
- Die Shannon-Expansion ordnet jedem Binären Entscheidungsdiagramm genau eine Boolesche Funktion zu. Da sich andersherum jede Boolesche Funktion durch und genau mit einem BDD darstellen lässt, sind BDDs kanonische Darstellung von Booleschen Funktionen. Ihre Verbindung zur Schaltalgebra ist somit evident.
- 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.

Binary Decision Tree

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 $\mathcal{N} := \mathcal{N}_{NT} \cup \mathcal{N}_T$.

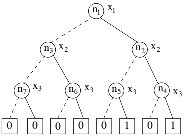


- Nodes are connected via 1- and 0-edges: $\longrightarrow \subseteq \mathcal{N}_{NT} \times \mathcal{N}$ and $- - \rightarrow \subseteq \mathcal{N}_{NT} \times \mathcal{N}$
- We read the tree from top to bottom, hence we can omitt the arrow heads
- Each inner node (circle) is assocoiated with a node label n_i and a variable x_j, e. g. $\mathtt{var}(n_6) = x_3$

- A dashed line leads to the 0-successor, the solid line to the 1-successor, e. g. $child_0(n_1) = n_3; \mathtt{child}_1(n_1) = n_2$
- Each terminal node is associated with a function value from $\mathbb{B} := \{0, 1\}$, e. g. $\mathtt{value}(t_1) = 0$

Binary Decision Tree: Variable Ordering

- 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 $\prec \subseteq \mathcal{V} \times \mathcal{V}$ and requires



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

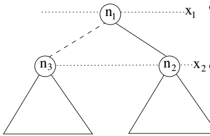
- What is the Boolean function represented by the BDT?
- What is the space complexity for representing Boolean functions with BDT?

Binary Decision Tree: Semantics

- Shannon expansion for Boolean functions:

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

  instead of the Boolean operators $(\neg, \vee, \wedge)$ 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 $f_1(x_1, \ldots, 1, x_{i+1}, \ldots, x_n)$ is denoted 1-cofactor of function f w. r. t. variable x_i.
- Function $f_0(x_1, \ldots, 0, x_{i+1}, \ldots, x_n)$ is denoted 0-cofactor of function f w. r. t. variable x_i.
- For the co-factors we also adapt the notation $f|_{x_i := b}$ 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:

$\longrightarrow$ 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 ($\mathcal{N} = \mathcal{N}_{NT} \cup \mathcal{N}_T$) as follows:

- for two terminal BDT-nodes $t, p \in \mathcal{N}_T$:

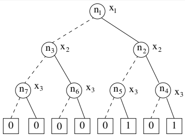
$$t \equiv p \Leftrightarrow \texttt{value}(t) = \texttt{value}(p)$$

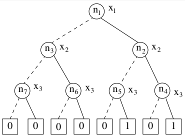
- for two non-terminal BDT-nodes $n, k \in \mathcal{N}_{NT}$:

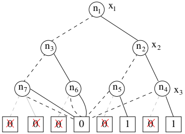
$$n \equiv k \Leftrightarrow \texttt{child}_0(n) \equiv \texttt{child}_0(k) \wedge \texttt{child}_1(n) \equiv \texttt{child}_1(k)$$

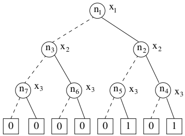
equivalent, i. e., $p \equiv t$ iff According to the above discussion each BDT-node represents a Boolean function.

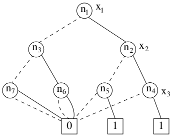
$\longrightarrow$ Question 1.2: How does this effect the size of the obtained graphs?

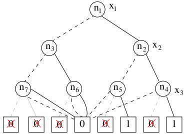


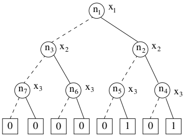


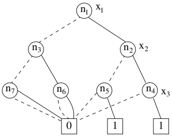


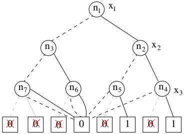

n1 x1
n3 x2 n2 x2
n7 x3 n6 x3 n5 x3 n4 x3
0 0 0 0 0 1 0 1

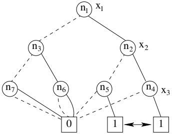

n1 x1
n3 n2 x2
n7 n6 n5 n4 x3
0 1 1

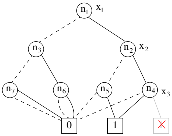

n1 x1
n3 n2 x2
n7 n6 n5 n4 x3
0 0 0 0 0 1 0 1

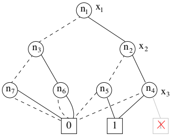

n1 x1
n3 x2 n2 x2
n7 x3 n6 x3 n5 x3 n4 x3
0 0 0 0 0 1 0 1

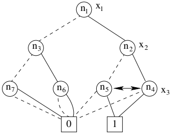

n1 x1
n3 n2 x2
n7 n6 n5 n4 x3
0 1 1

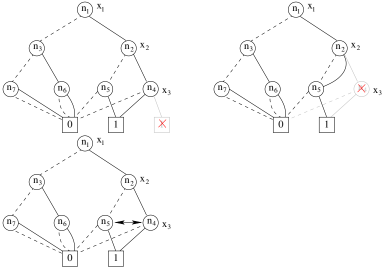

n1 x1
n3 n2 x2
n7 n6 n5 n4 x3
0 0 0 0 0 1 0 1


n1 x1
n3 n2 x2
n7 n6 n5 n4 x3
0 1 1

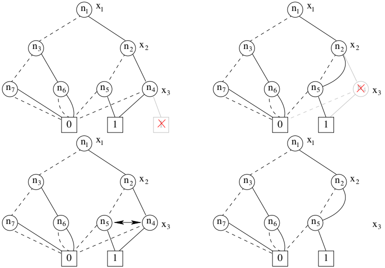

n1 x1
n3 n2 x2
n7 n6 n5 n4 x3
0 1


n1 x1
n3 n2 x2
n7 n6 n5 n4 x3
0 1

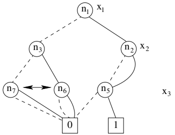

n1 x1
n3 n2 x2
n7 n6 n5 n4 x3
0 1



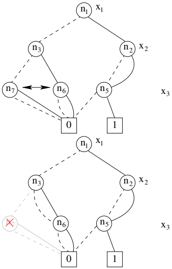
n1 x1
n3
n2 x2
n7
n6
n5
n4 x3
0
1



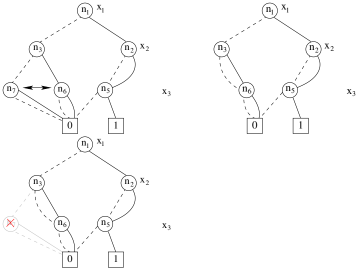
n1 x1
n3
n2 x2
n7
n6
n5
n4 x3
0
1

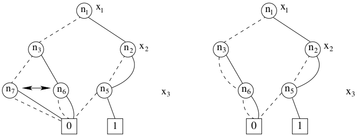


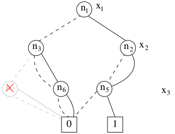
n1 x1
n3
n2 x2
n7
n6
n5
x3
0
1



n1 x1
n3
n2 x2
n7
n6
n5
x3
0
1

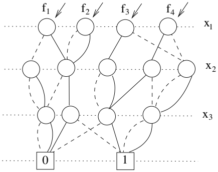






- The reduction is applied on the fly, i. e., each allocated node is unique.
- Hence application of an a posteriori reduction not necessary.
- As we will see later, uniqueness of nodes is not only a key to memory efficiency but also to run-time efficiency w. r. t. the manipulation of BDDs.

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



⟶ Question 1.3: Can we do more, e. g. apply Shannon for function f_3?

A node $n \in \mathcal{N}_{NT}$ is denoted **don't-care (dnc)** node iff

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

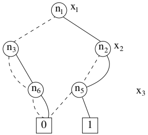
As shown by the example the Shannon-expansion yields, that such nodes can safely be ommited once one allocates BDDs.

Binary Decision Diagram: Don't care nodes

A node $n \in \mathcal{N}_{NT}$ is denoted **don't-care (dnc)** node iff

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

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

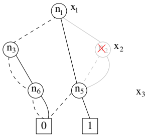


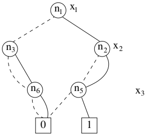
Binary Decision Diagram: Don't care nodes

A node $n \in \mathcal{N}_{NT}$ is denoted **don't-care (dnc)** node iff

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

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



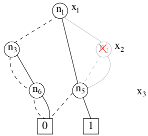


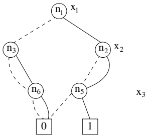
Binary Decision Diagram: Don't care nodes

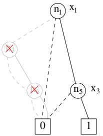
A node $n \in \mathcal{N}_{NT}$ is denoted **don't-care (dnc)** node iff

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

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





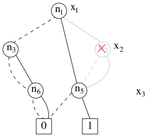


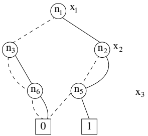
Binary Decision Diagram: Don't care nodes

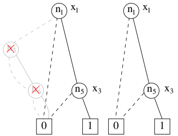
A node $n \in \mathcal{N}_{NT}$ is denoted **don't-care (dnc)** node iff

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

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







A reduced ordered Binary Decision Diagram B $< \mathcal{V}, \prec >$ is a 5-tuple $\{\mathcal{N}, \texttt{value}, \texttt{var}, \texttt{child}_1, \texttt{child}_0\}$ where

1. $\mathcal{V}$ is a finite and non-empty set of boolean variables with the fixed ordering relation $\prec \subseteq \mathcal{V} \times \mathcal{V}$ defined one.
2. $\mathcal{N} = \mathcal{N}_T \cup \mathcal{N}_{NT}$ is a finite non-empty set of nodes, consisting of the set of terminal nodes $\mathcal{N}_T$ and non-terminal nodes $\mathcal{N}_{NT}$, with $\mathcal{N}_T \cap \mathcal{N}_{NT} = \emptyset$.
3. The following functions are defined:
   1. the value-returning function $\texttt{value} : \mathcal{N}_T \mapsto \mathbb{B}$ for each terminal node,
   2. the variable-returning function $\texttt{var} : \mathcal{N}_{NT} \mapsto \mathcal{V}$ for each non-terminal node,
   3. the child node-returning functions $\texttt{child}_0, \texttt{child}_1 : \mathcal{N}_{NT} \mapsto \mathcal{N}$ for each non-terminal node, and
   4. the root node-returning function $\texttt{getRoot} : \mathrm{B} \mapsto \mathcal{N}$.

Binary Decision Diagrams: Formal Definition

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

$$\begin{aligned} &\forall u \in \mathcal{N}_{NT} : \\ &\quad \texttt{child}_1(u) \in \mathcal{N}_{NT} : \texttt{var}(\texttt{child}_1(u)) \succ \texttt{var}(u) \\ &\quad \texttt{child}_0(u) \in \mathcal{N}_{NT} : \texttt{var}(\texttt{child}_0(u)) \succ \texttt{var}(u). \end{aligned}$$

2. A BDD is denoted reduced *iff* the following conditions apply:
   (a) Isomorphism rule: No isomorphic nodes; i.e.
      (i) Non-terminal case: $\forall n, m \in \mathcal{N}_{NT}$:
      $$n \neq m \Rightarrow (\texttt{var}(n) \neq \texttt{var}(m) \vee \texttt{child}_1(n) \neq \texttt{child}_1(m) \vee \texttt{child}_0(n) \neq \texttt{child}_0(m))$$
      (ii) Terminal case: $\forall n, m \in \mathcal{N}_T : n \neq m \Rightarrow (\texttt{value}(n) \neq \texttt{value}(m))$

   (b) Dnc-rule: No don't care nodes: $\nexists u \in \mathcal{N}_{NT} : \texttt{child}_0(u) = \texttt{child}_1(u)$.

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

$$f \not\equiv g \Leftrightarrow \mathrm{B}_f \not\equiv \mathrm{B}_g$$

$\longrightarrow$ Question 1.4: Why can equivalenz testing be done in constant time?

- Consider the following two Boolean functions:

$$\begin{aligned} f &:= \neg dab + \neg ad\neg c + abd + \neg a\neg c\neg d \\ g &:= \neg a\neg cb + cba + \neg b\neg a\neg c + a\neg bc \end{aligned}$$

$\longrightarrow$ Question 1.5: Are f and g equivalent? Please justify by making use of BDDs

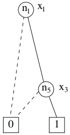
$\longrightarrow$ Excursion 1.1: Proof of canonicity (on the black board)

- 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.

$$\begin{aligned} \neg f &\rightsquigarrow \texttt{Negate}(\mathrm{B}_f) \\ f + g &\rightsquigarrow \texttt{Plus}(\mathrm{B}_f, \mathrm{B}_g) \\ f \cdot g &\rightsquigarrow \texttt{Mult}(\mathrm{B}_f, \mathrm{B}_g) \end{aligned}$$

BDD-based algorithms: Negation

```
Negate(node n)
(0) IF n ∈ 𝒩_T THEN
        RETURN(makeTerminal(1 − value(n)))
(1) ELSE
(2)     node t := Negate(child_1(n))
(3)     node e := Negate(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

BDD-based algorithms: Apply

A binary operator op $\in \{+, \times, \ldots\}$ can be applied to BDDs by means of Bryant's Apply algorithm.

```
APPLY(op, node n, node m)
(0)     node e, t, res

Reached terminal nodes, end of recursion
(1)     IF n, m ∈ 𝒩_T THEN
(2)         int v := value(n) op value(m)
(3)         RETURN(makeTerminal(v))

Check op cache if result is already known
(4)     res := cacheLookup(op, n, m)
(5)     IF res ≠ ϵ THEN RETURN(res)

Depending on the node-labelling variables
                              branch into recursion
(6)     IF var(n) = var(m) THEN
(7)         v := var(n)
(8)         e := APPLY(op, child_0(n), child_0(m))
(9)         t := APPLY(op, child_1(n), child_1(m))

(10)    ELSE IF var(n) ≺ var(m) THEN
(11)        v := var(n)
(12)        e := APPLY(op, child_0(n), m)
(13)        t := APPLY(op, child_1(n), m)

(14)    ELSE
(15)        v := var(m);
(16)        e := APPLY(op, n, child_0(m))
(17)        t := APPLY(op, n, child_1(m))

Allocate new node, unique and non-dnc-node
(18)    IF t = e THEN RETURN(t)
(19)    ELSE
(20)        res := RETURN(makeNode(v, t, e))

Insert result into op cache and terminate recursion
(19)    cacheInsert(op, n, m, res)
(20)    RETURN(res)
```

⟶ 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 $f_1 \wedge f_2$ and give the resulting BDD.

BDD: Algorithmic manipulation

Besides the APPLY-algorithm, which is the most important one other algorithms have been developped. Let f be a n-ary Boolean function and let B^f be its BDD, in the following we will employ the following operation and their resp. BDD-based implementations:

- $f(x_1, \ldots, x_n)|_{x_i = b} \rightsquigarrow \mathtt{RESTRICT}(B^f, b)$ with $b \in \mathbb{B}$
- Quantification:
  1. Existential quantification:
     $$\exists x_i : f(\vec{x}) \Leftrightarrow f|_{x_i=1} \times f|_{x_i=0} \rightsquigarrow \mathtt{ABSTRACT}(B, x_i, \mathtt{Mult})$$
  2. Universal quantification:
     $$\forall x_i : f(\vec{x}) \Leftrightarrow f|_{x_i=0} + f|_{x_i=1} \rightsquigarrow \mathtt{ABSTRACT}(B, x_i, \mathtt{Plus})$$
- Relabeling: $[x \mapsto y]f \rightsquigarrow B^f\{y \leftarrow x\}$, each occurence of variable x is replaced by variable y: $(f|_{x=1} \times (g(y) = y)) + (f|_{x=1} \times (g(y) = y))$

⟶ Example 1.2: BDDs

BDD-based approaches for the Verification of systems

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 rechability set generation, especially in case of Petri nets
- Symbolic CTL model checking