

COMP6463: Temporal Logic and Model Checking

4. Symbolic Model Checking

Michael Norrish

Canberra Research Lab., NICTA

Semester 1, 2009



NICTA

Outline

- 1 Introduction
- 2 Interlude: Binary Decision Diagrams
- 3 Modelling Systems with BDDs
- 4 CTL Checking Algorithm with BDDs

What We've Done Already

Modal Logic

- Restricted, **local**, quantifiers over relational structures.

CTL

- A modal logic with quantifications over future events and branching paths.

Explicit State Model Checking

- The general idea (example: JPF), and
- In **CTL** case, how to determine if a finite system satisfies an arbitrary **CTL** property.

The Fundamental Problem with Model Checking

State Space Explosion



State Space Explosion

Realistic systems have many states.

—Understatement of the Century

Much model checking research is all about finding clever ways to

- reduce systems to “equivalent” systems with fewer states
- represent states/systems more efficiently
- check only the states that “matter”

Model Checking Capabilities

- Early (= early 1980s) systems able to check systems of 10^4 to 10^5 states (at ~ 100 states per second)
- By using OBDDs (late 1980s), systems of up to 10^{20} states
- More recently (according to Clarke), systems of up to 10^{120} states

Symbolic Model Checking

Due to Ken McMillan (now at Cadence) in his 1992 PhD (at CMU)

*Symbolic Model Checking:
An approach to the state explosion problem*

Works on the “represent states/systems more efficiently” angle.

Idea is to represent systems “symbolically”.

(Contrast [Explicit State Model Checking](#).)

Binary Decision Diagrams

The symbolic representation for a system will be a (set of) **binary decision diagram(s)**.

Strictly, we should say **ordered binary decision diagrams**.

- With an ordering imposed, get important **canonicity** properties

OBDDs were invented by Randy Bryant in the mid 80s.

Specification

A BDD is a representation of a function

- from n boolean **inputs**, to
- one boolean **output**

Alternatively, a BDD is a representation of a formula of propositional logic.

This formula takes as inputs the values of variables p , q and r and returns true or false:

$$(p \vee \neg r) \wedge (q \vee r) \wedge (\neg p \vee \neg r)$$

Running Example

The two-bit comparator (from Clarke *et al*):

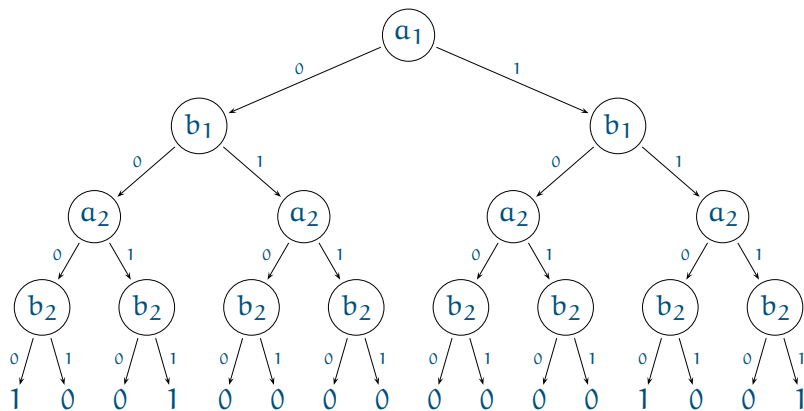
Four inputs: a_1, a_2, b_1, b_2

Output: True iff $a_1 = b_1$ and $a_2 = b_2$

Binary Decision Trees

A tree where each node tests the value of a variable.

Alternatively, a big if-then-else expression.

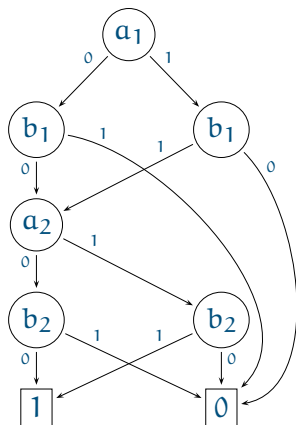


Making Trees Efficient

A binary decision tree is exponential in the number of variables.

But turn a tree into a DAG and it can become dramatically more efficient:

- Remove redundant checks
- Share isomorphic sub-trees



Defining the OBDD

A “binary decision DAG” is an **ordered binary decision diagram** if it is

- maximally reduced (as on previous slide); and
- variables from root to leaf appear in the same order (when they appear at all)

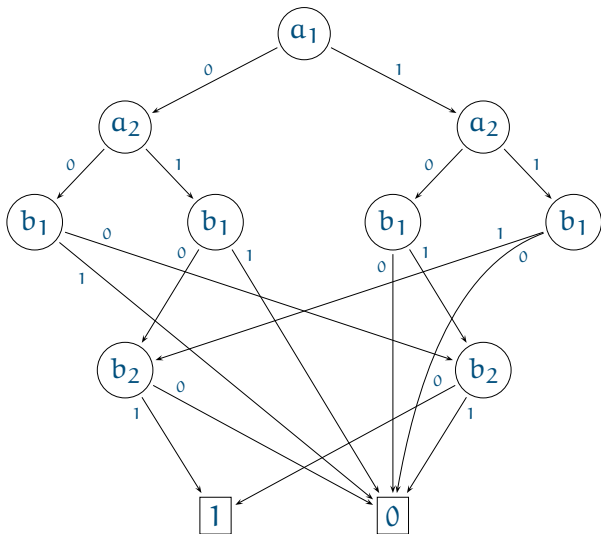
Canonicity:

*Having picked an order for the variables, there is **only one** OBDD for any given formula/function.*

Variable Order Makes a Difference

Before, we used
 a_1, b_1, a_2, b_2

If we use
 a_1, a_2, b_1, b_2
performance is much worse.



More on Variable Orders

Finding a formula's optimal variable order is “infeasible”

- even determining if a given order is optimal is NP-complete

Picking orders that put “related” variables together seems to work quite well.

Some formulas give exponentially sized BDDs for all possible orders.

- For example: multiplication circuits

BDD Implementations

Provide efficient implementations of normal boolean operators (\wedge , \vee etc) over BDDs

- Clever games are played with caches
- Cost is linear in product of formula sizes

Provide **constant-time equality checks** on formulas.

Provide tricks like **dynamic variable reordering** in order to heuristically make BDDs smaller (and faster!)

Example: Conjunction for BDDs

Remember that a BDD is “just” an if-then-else expression with variables as guards.

Lemma 1:

$$\begin{aligned}(\text{if } p \text{ then } \phi_1 \text{ else } \phi_2) \wedge \psi &\equiv \\ &(\text{if } p \text{ then } (\phi_1 \wedge \psi) \text{ else } (\phi_2 \wedge \psi))\end{aligned}$$

Use when p is earlier in the variable order than ψ 's top variable.

Lemma 2:

$$\begin{aligned}(\text{if } p \text{ then } \phi_1 \text{ else } \phi_2) \wedge (\text{if } p \text{ then } \psi_1 \text{ else } \psi_2) &\equiv \\ &(\text{if } p \text{ then } (\phi_1 \wedge \psi_1) \text{ else } (\phi_2 \wedge \psi_2))\end{aligned}$$

For when the variables are the same.

BDDs: Some Extra Notes

Building even a small BDD can require many resources

- e.g., build the BDD of ϕ (the description of a huge multiplier); then conjoin this with $\neg\phi$; result will be \perp

BDDs not necessarily ideal for solving SAT.

Ideal for equivalence checking

- e.g., in combinational circuit optimisation

Model Checking Operations on BDDs

It's also important to be able to:

Substitute: Calculate $\phi[v := b]$ with $b \in \{\top, \perp\}$. (Easy tree traversal.)

Rename: Calculate $\phi[v_1 := v_2]$.

Turn this into (if v_2 then $\phi[v_1 := \top]$ else $\phi[v_1 := \perp]$)

Quantify: Calculate $\exists v. \phi$.

This is equivalent to $\phi[v := \top] \vee \phi[v := \perp]$

Representing a Kripke Structure with a BDD

A BDD represents a function: $f : 2^n \rightarrow 2$

This can equally be seen as a relation on n boolean variables.

If $n = 2m$, a binary relation on tuples of m variables:

$$f : 2^m \rightarrow 2^m \rightarrow 2$$

And we know finite things can be encoded into fixed size vectors of booleans. . .

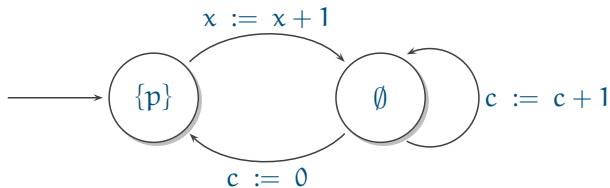
BDDs Can Do it All

BDDs can encode:

- relations over states;
- sets of states (a set is a unary relation);
- and fairness constraints (which are just sets of states)

Encoding Example

Let x and c be three-bit counters, and p our one boolean propositional variable.



Need 7 boolean variables to capture a state:

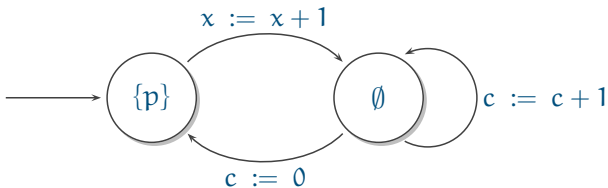
$$x_0, x_1, x_2, c_0, c_1, c_2, p$$

Lower arrow (from right to left) is captured by clause:

$$\begin{aligned}x'_0 &= x_0 \wedge x'_1 = x_1 \wedge x'_2 = x_2 \wedge \\c'_0 &= \perp \wedge c'_1 = \perp \wedge c'_2 = \perp \wedge \\p &= \perp \wedge p' = \top\end{aligned}$$

Encoding Example

Let x and c be three-bit counters, and p our one boolean propositional variable.



Lower arrow (from right to left) is captured by clause:

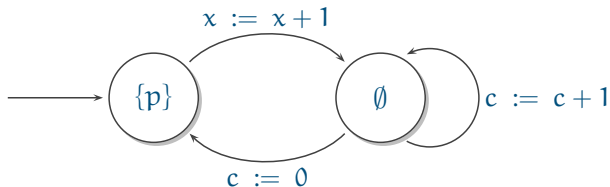
$$\begin{aligned}x'_0 &= x_0 \wedge x'_1 = x_1 \wedge x'_2 = x_2 \wedge \\c'_0 &= \perp \wedge c'_1 = \perp \wedge c'_2 = \perp \wedge \\p &= \perp \wedge p' = \top\end{aligned}$$

Other clauses more complicated because of the need to encode arithmetic.

(They would be additional disjuncts. . .)

Encoding Example

Let x and c be three-bit counters, and p our one boolean propositional variable.



Should be able to prove

$$AG (\neg p \Rightarrow EX (c = 0))$$

(Note how I've made our CTL language more expressive.)

Symbolic CTL Checking

The checking procedure for a CTL formula will

- take as **input** the Kripke structure \mathcal{R} (a BDD);
- take as **input** the CTL formula ϕ ;
- return as **output** the set of states satisfying ϕ (another BDD)

Symbolic CTL Checking—Easy Cases

The set of states satisfying

- p (a propositional variable) is the BDD for p
- $\neg\phi$ (a negation) is the negation of BDD for ϕ
- $\phi_1 \vee \phi_2$ is the disjunction of the BDDs for ϕ_1 and ϕ_2

Set-based Computation of CTL Modalities

Have a BDD R over variables $v_1 \dots v_n$ and $v'_1 \dots v'_n$.

To compute $EX \phi$:

- 1 Recursively convert ϕ to a BDD f .

f is a BDD over variables $v_1 \dots v_n$ that identifies those states where ϕ is true.

- 2 Result is: $\exists v'. f[v := v'] \wedge R$

(Here v and v' stand for **vectors** of variables.)

Result is true of $v_1 \dots v_n$ if there is a reachable successor state $(v'_1 \dots v'_n)$ satisfying f .

Set-based Computation of Until

Define

$$A[\phi_1 \cup \phi_2]_0 \equiv \phi_2$$

$$A[\phi_1 \cup \phi_2]_{(i+1)} \equiv A[\phi_1 \cup \phi_2]_i \vee (\phi_1 \wedge AX(A[\phi_1 \cup \phi_2]_i))$$

Then, $A[\phi_1 \cup \phi_2]_i$ is true of a state

- if ϕ_2 comes true on every path within i steps, and
- if ϕ_1 is true on those paths up until the point when ϕ_2 becomes true.

So, $A[\phi_1 \cup \phi_2] \equiv \exists i. A[\phi_1 \cup \phi_2]_i$

(This does rely on system being finitely branching.)

Set-based Computation of Until

Define

$$A[\phi_1 \cup \phi_2]_0 \equiv \phi_2$$

$$A[\phi_1 \cup \phi_2]_{(i+1)} \equiv A[\phi_1 \cup \phi_2]_i \vee (\phi_1 \wedge AX(A[\phi_1 \cup \phi_2]_i))$$

Can (recursively) calculate BDDs for $A[\phi_1 \cup \phi_2]_i$

Critically: $A[\phi_1 \cup \phi_2]_i \subseteq A[\phi_1 \cup \phi_2]_{i+1}$

Because there are only **finitely many states** in the system:

- there will be an i such that $A[\phi_1 \cup \phi_2]_i = A[\phi_1 \cup \phi_2]_{i+1}$

Set-based Computation of Until

Define

$$A[\phi_1 \cup \phi_2]_0 \equiv \phi_2$$

$$A[\phi_1 \cup \phi_2]_{(i+1)} \equiv A[\phi_1 \cup \phi_2]_i \vee (\phi_1 \wedge AX(A[\phi_1 \cup \phi_2]_i))$$

Successively compute BDDs

$$A[\phi_1 \cup \phi_2]_0, A[\phi_1 \cup \phi_2]_1, A[\phi_1 \cup \phi_2]_2, \dots$$

until the fixed-point is reached.

- This BDD will then be the BDD for $A[\phi_1 \cup \phi_2]$

(Similarly for $E[\phi_1 \cup \phi_2] \dots$)

Model Checking and Initial States

Initial States are very important.

Let

I = the initial states of the system

S = the set of states satisfying ϕ

We may want, for example, $I \subseteq S$

- if ϕ is some good safety (invariant property)

Or, we may not care if an unreachable state doesn't satisfy ϕ .

Reachability Analysis

The reachable states in a finite system are a finite set.

The reachable states can be represented by a BDD

$$Reach_0 = I$$

$$Reach_{i+1} = Reach_i \vee \exists v_0. Reach_i[v := v_0] \wedge R[v := v_0, v' := v]$$

(Each step calculates the **image** of R ; the **EX** calculation calculates the **pre-image**.)

Precomputing reachable states helps with (common) CTL queries of the form

$$I \Rightarrow AG \phi$$

Role of BDDs in Summary

BDDs represent both

- transition relation; and
- formulas (sets of states)

BDDs support important operations:

- quantification over variables (for X modality)
- fast equality checks (for fixed-point detection)

Summary

- The fundamental model checking problem is the **state space explosion**
- **BDDs** provide efficient representations of sets of boolean variables
- BDDs can be used to **check CTL formulas**
 - states (and transition relations) are no longer explicit, but represented **symbolically** with BDDs