UNIVERSITY OF TWENTE.

Formal Methods & Tools.

Binary Decision Diagrams for Reachability Analysis

Jaco van de Pol

24 April 2018, SofSci

Goal and Topics

Goal of Theory Lectures:

- Model Checking: verify properties of concurrent systems automatically based on enumerative methods
- Main problem: size of the state space (large graph)
- Focus on Algorithms and Data structures for Model Checking

Goal and Topics

Goal of Theory Lectures:

- Model Checking: verify properties of concurrent systems automatically based on enumerative methods
- Main problem: size of the state space (large graph)
- Focus on Algorithms and Data structures for Model Checking

Planning of the Theory Block

- 24/4: Binary Decision Diagrams (concise representation)
- 1/5: CTL model checking (Computation Tree Logic)
- 8/5: LTL model checking (Linear Time Logic)
- 15/5: High-Performance Model Checking (LTSmin tool)

Practical Assignment

The practical assignment is carried out in self-formed pairs.

Provided (in C or C++):

- BDD package, in particular Sylvan
- Parser for Petri Nets from MCC competition

Assignment, including a competition:

1. Build a symbolic reachability checker (intermediate)
2. Build a symbolic CTL (LTL) model checker (end goal)

Practical Assignment

The practical assignment is carried out in self-formed pairs.

Provided (in C or C++):

- BDD package, in particular Sylvan
- Parser for Petri Nets from MCC competition

Assignment, including a competition:

1. Build a symbolic reachability checker (intermediate)
2. Build a symbolic CTL (LTL) model checker (end goal)

Planning Practical Assignment

- Kickoff Practical Assignment (2 weeks)
- 26/4, 3/5 and 17/5: Practical sessions on by Jeroen Meijer

Material for Today

Papers/Tutorials (online available)

- H.R. Andersen, *An Introduction to Binary Decision Diagrams* (obligatory: Section 1-5)
- R.E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams* (classical, recommended)

Material for Today

Papers/Tutorials (online available)

- H.R. Andersen, *An Introduction to Binary Decision Diagrams* (obligatory: Section 1-5)
- R.E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams* (classical, recommended)

Books (if you want to read more)

- Clarke, Grumberg, Peled, *Model Checking*, Chapters 5, 6
- Baier, Katoen, *Principles of Model Checking*, Chapter 6.7

Material for Today

Papers/Tutorials (online available)

- H.R. Andersen, *An Introduction to Binary Decision Diagrams* (obligatory: Section 1-5)
- R.E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams* (classical, recommended)

Books (if you want to read more)

- Clarke, Grumberg, Peled, *Model Checking*, Chapters 5, 6
- Baier, Katoen, *Principles of Model Checking*, Chapter 6.7

Tools

- BDD-packages: BuDDy, CuDD, Java(B)DD, Sylvan
- Sylvan: `https://github.com/utwente-fmt/sylvan`
- LTSmin: `http://fmt.cs.utwente.nl/tools/ltsmin/` and `https://github.com/utwente-fmt/ltsmin`

Table of Contents

Reachability Problems

The Reachability Problem

- Given a graph $G = (V, R)$, initial states $I \subseteq V$ and goal/error states $F \subseteq V$, check if there is a path from I to F in G.
- Typically, the graph is given implicitly, as a program/specification

Reachability Problems

The Reachability Problem

- Given a graph $G = (V, R)$, initial states $I \subseteq V$ and goal/error states $F \subseteq V$, check if there is a path from I to F in G.
- Typically, the graph is given implicitly, as a program/specification

Examples

- Find assertion violations in multi-core software
- Find safety risks in Railway Interlockings
- Find solutions to games/puzzles, e.g. Sokoban

Reasons for State Space Explosion

Concurrency

- System of n components, each can be in m states
- The total state space may consist of m^n states.
- Example: Railway safety systems (signals, points, tracks)

Reasons for State Space Explosion

Concurrency

- System of n components, each can be in m states
- The total state space may consist of m^n states.
- Example: Railway safety systems (signals, points, tracks)

Data variables

- Given n different variables, each may take m values
- Potential number of different state vectors: m^n
- Example: model checking software, rather than models

Reasons for State Space Explosion

Concurrency

- System of n components, each can be in m states
- The total state space may consist of m^n states.
- Example: Railway safety systems (signals, points, tracks)

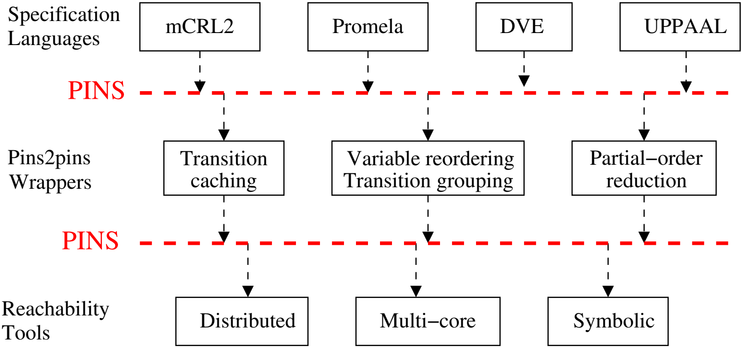
Data variables

- Given n different variables, each may take m values
- Potential number of different state vectors: m^n
- Example: model checking software, rather than models

How to handle $> 10^{100}$ states??

- No way to explicitly enumerate that many states
- Partial Order Reduction: Skip certain states systematically
- Symbolic model checking: Treat sets of states simultaneously

LTSmin Toolset (University of Twente)



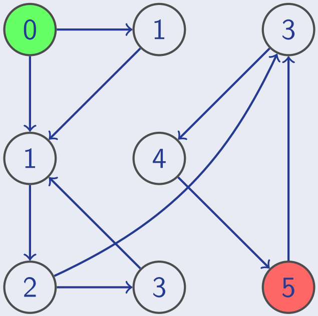
LTSmin unique selling points

- Multiple languages, Multiple engines, Generic reductions
- Analysis: bisimulation reduction, LTL and CTL* model checking, mu-calculus

Table of Contents

Breadth First Search, example

BFS traversal



- Eventually all states will be visited
- Shortest path will be detected

Breadth First Search

Explicit-state BFS

```
check that init ∉ Error
Queue := [init]
Visited := {init}
while Queue ≠ [] do
    pick s from front of Queue
    for all t with s → t do
        if t ∉ Visited then
            check that t ∉ Error
            put t to the end of Queue
            add t to Visited
        end if
    end for
end while
```

Breadth First Search

Explicit-state BFS

```
check that init ∉ Error
Queue := [init]
Visited := {init}
while Queue ≠ [] do
    pick s from front of Queue
    for all t with s → t do
        if t ∉ Visited then
            check that t ∉ Error
            put t to the end of Queue
            add t to Visited
        end if
    end for
end while
```

Set-based BFS (variant 1)

```
Vis := Cur := {init}
while Cur ≠ ∅ do
    check Cur ∩ Error = ∅
    Cur := Next(Cur, →) \ Vis
    Vis := Vis ∪ Cur
end while
```

Breadth First Search

Explicit-state BFS

```
check that init ∉ Error
Queue := [init]
Visited := {init}
while Queue ≠ [] do
    pick s from front of Queue
    for all t with s → t do
        if t ∉ Visited then
            check that t ∉ Error
            put t to the end of Queue
            add t to Visited
        end if
    end for
end while
```

Set operations: $\cap$, $\cup$, $\overline{S}$, $=$, Next

Set-based BFS (variant 1)

```
Vis := Cur := {init}
while Cur ≠ ∅ do
    check Cur ∩ Error = ∅
    Cur := Next(Cur, →) \ Vis
    Vis := Vis ∪ Cur
end while
```

Set-based BFS (variant 2)

```
V_old := ∅
V_new := {init}
while V_old ≠ V_new do
    V_old := V_new
    V_new := V_old ∪ Next(V_old, →)
end while
```

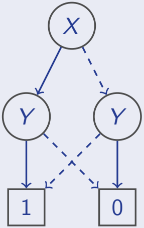
Table of Contents

Binary Decision Diagrams

Binary Decision Diagram

- A Binary Decision Diagram is a directed acyclic graph
- Its internal nodes are ordered, binary (called low, high)
- Its internal nodes are labeled by variables
- Its leaves are labeled by 0 or 1

Example

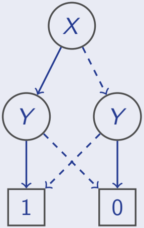


Binary Decision Diagrams

Binary Decision Diagram

- A Binary Decision Diagram is a directed acyclic graph
- Its internal nodes are ordered, binary (called low, high)
- Its internal nodes are labeled by variables
- Its leaves are labeled by 0 or 1

Example



Conventions

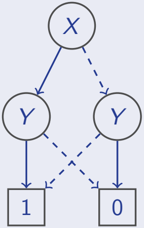
- Internal nodes are drawn as circles
- High edges are drawn solid
- Low edges are drawn dashed
- Leaves are drawn as boxes, with 0 or 1
- "If X is true, then high, else low branch"
- Formula on the left:

Binary Decision Diagrams

Binary Decision Diagram

- A Binary Decision Diagram is a directed acyclic graph
- Its internal nodes are ordered, binary (called low, high)
- Its internal nodes are labeled by variables
- Its leaves are labeled by 0 or 1

Example



Conventions

- Internal nodes are drawn as circles
- High edges are drawn solid
- Low edges are drawn dashed
- Leaves are drawn as boxes, with 0 or 1
- "If X is true, then high, else low branch"
- Formula on the left: $X \Leftrightarrow Y$

How to interpret a BDD?

Boolean Functions

- Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be Boolean variables
- A valuation is a function $\mathcal{X} \to \{0, 1\}$
- A BDD represents a set of valuations
  - all valuations that lead from the root to leaf 1 are in the set
  - valuations that lead from the root to leaf 0 are not in the set
- Equivalently, a BDD represents a function $\{0, 1\}^n \to \{0, 1\}$

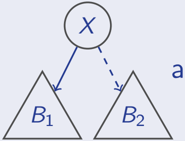
How to interpret a BDD?

Boolean Functions

- Let $\mathcal{X} = \{x_1, \ldots, x_n\}$ be Boolean variables
- A valuation is a function $\mathcal{X} \to \{0, 1\}$
- A BDD represents a set of valuations
  - all valuations that lead from the root to leaf 1 are in the set
  - valuations that lead from the root to leaf 0 are not in the set
- Equivalently, a BDD represents a function $\{0, 1\}^n \to \{0, 1\}$

Hint

You can read the BDD



as one of:

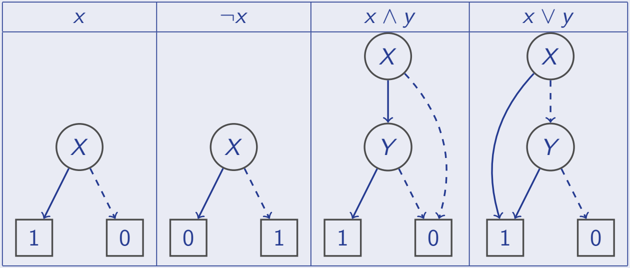
- If X then B_1 else B_2. Notation: $X \to B_1, B_2$
- $(X \wedge B_1) \vee (\neg X \wedge B_2)$ (aka Shannon's expansion).

Examples

Basic Boolean Connectives

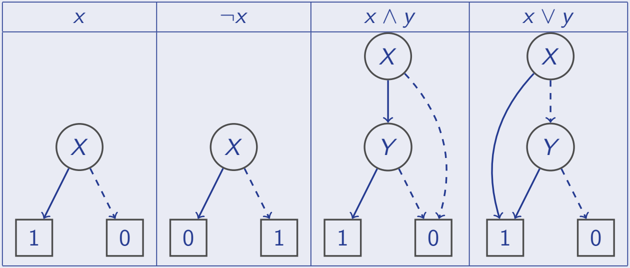
Examples

Basic Boolean Connectives



Examples

Basic Boolean Connectives

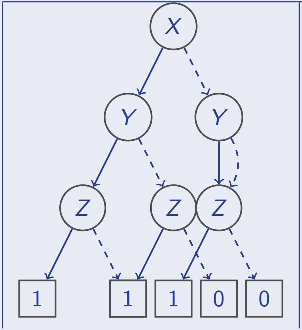


Propositional logic formulas

- Apparently, BDDs form an alternative to proposition logic.
- Recall negation $\neg$ and the binary connectives: $\wedge$, $\vee$, $\Rightarrow$, $\Leftrightarrow$
- How many binary operators are possible? ...sufficient?
- Introduce a ternary operator: $x \rightarrow s, t$; ..... sufficient basis!

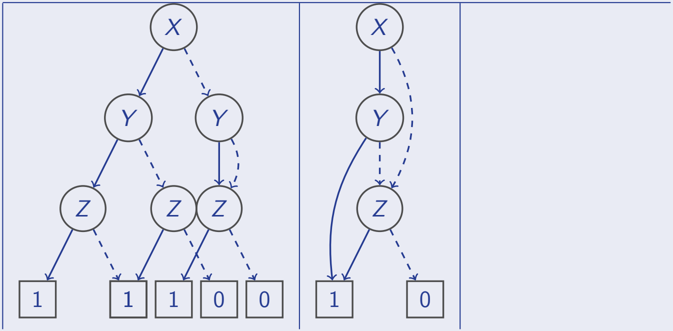
More Examples

Three times: $(x \wedge y) \vee z$



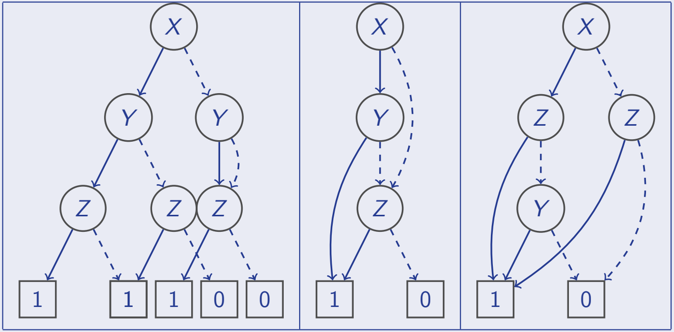
More Examples

Three times: $(x \wedge y) \vee z$



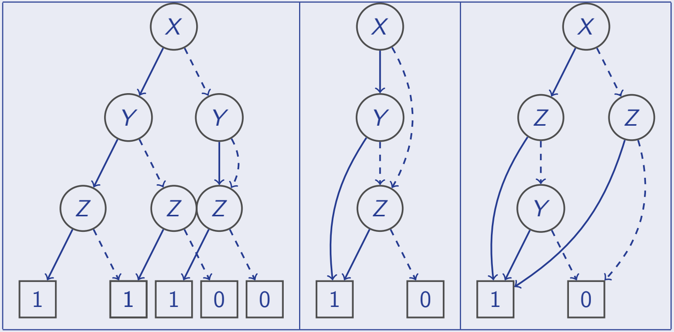
More Examples

Three times: $(x \wedge y) \vee z$



More Examples

Three times: $(x \wedge y) \vee z$

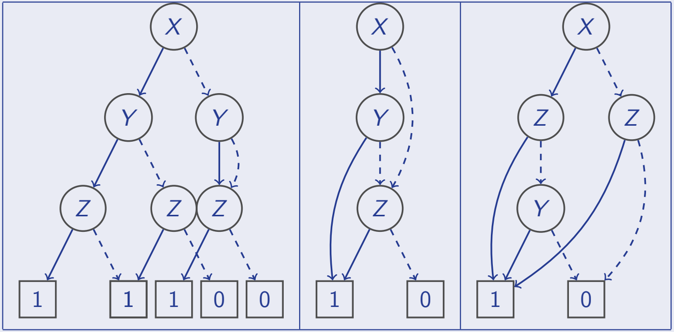


Reduced BDDs:

- no duplicate nodes
- no redundant tests

More Examples

Three times: $(x \wedge y) \vee z$



Reduced BDDs:

- no duplicate nodes
- no redundant tests

Ordered BDDs:

- The order of the vars is fixed
- The order impacts BDD size

Reduced Ordered BDDs (ROBDD = OBDD)

Reduced BDDs

A BDD is called reduced iff:

- No duplicate leafs:
  There is at most one leaf with label 0 and one with label 1.
- No duplicate nodes: For all nodes v, w, if $var(v) = var(w)$, $low(v) = low(w)$ and $high(v) = high(w)$, then $v = w$.
- No redundant tests: For all nodes v, $low(v) \neq high(v)$.

Reduced Ordered BDDs (ROBDD = OBDD)

Reduced BDDs

A BDD is called reduced iff:

- No duplicate leafs:
  There is at most one leaf with label 0 and one with label 1.
- No duplicate nodes: For all nodes v, w, if $var(v) = var(w)$, $low(v) = low(w)$ and $high(v) = high(w)$, then $v = w$.
- No redundant tests: For all nodes v, $low(v) \neq high(v)$.

Ordered BDDs

A BDD is called ordered iff

- there exists an ordering $x_1 < x_2 < \cdots < x_n$, such that
- for all nodes v in the BDD, $var(v) < var(low(v))$ and $var(v) < var(high(v))$

From Boolean Functions to OBDDs

Simple operations: Restriction and Renaming

- The operation $B[x := c]$ for $c \in \{0, 1\}$ is called restriction
- Renaming replaces variable names, as in $B[x/x']$.

From Boolean Functions to OBDDs

Simple operations: Restriction and Renaming

- The operation $B[x := c]$ for $c \in \{0, 1\}$ is called restriction
- Renaming replaces variable names, as in $B[x/x']$.

Three ways of transforming formulas to OBDDs:

- Top down. (insightful; 2^n many recursive calls)
  - Recursively, apply so-called Shannon's expansion:
  - Find the smallest variable x in formula P.
  - Replace P by $x \to P[x := 0], P[x := 1]$; repeat on subterms.

From Boolean Functions to OBDDs

Simple operations: Restriction and Renaming

- The operation $B[x := c]$ for $c \in \{0, 1\}$ is called restriction
- Renaming replaces variable names, as in $B[x/x']$.

Three ways of transforming formulas to OBDDs:

- Top down. (insightful; 2^n many recursive calls)
  - Recursively, apply so-called Shannon's expansion:
  - Find the smallest variable x in formula P.
  - Replace P by $x \to P[x := 0], P[x := 1]$; repeat on subterms.
- Stepwise. (nice proof; not very practical)
  - Replace all Boolean connectives by appropriate $x \to y, z$
  - Reduce and order the BDD by stepwise transformation

From Boolean Functions to OBDDs

Simple operations: Restriction and Renaming

- The operation $B[x := c]$ for $c \in \{0,1\}$ is called restriction
- Renaming replaces variable names, as in $B[x/x']$.

Three ways of transforming formulas to OBDDs:

- Top down. (insightful; 2^n many recursive calls)
  - Recursively, apply so-called Shannon's expansion:
  - Find the smallest variable x in formula P.
  - Replace P by $x \rightarrow P[x := 0], P[x := 1]$; repeat on subterms.
- Stepwise. (nice proof; not very practical)
  - Replace all Boolean connectives by appropriate $x \rightarrow y, z$
  - Reduce and order the BDD by stepwise transformation
- Bottom up. (this is what BDD packages do)
  - Start with converting subterms to OBDD data structure
  - Continue by defining AND, OR, NEG directly on OBDDs

Stepwise transformation from BDD to (R)OBDD

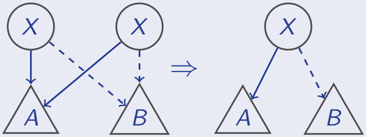
A BDD can (in principle) be transformed to an OBDD by repeated application of the following transformation rules:

Stepwise transformation from BDD to (R)OBDD

A BDD can (in principle) be transformed to an OBDD by repeated application of the following transformation rules:

Stepwise reduction

- Eliminate duplicate nodes:



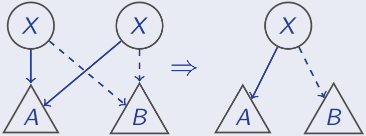
- Eliminate redundant tests:

Stepwise transformation from BDD to (R)OBDD

A BDD can (in principle) be transformed to an OBDD by repeated application of the following transformation rules:

Stepwise reduction

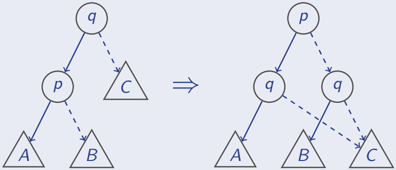
- Eliminate duplicate nodes:



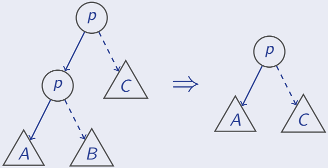
- Eliminate redundant tests:

Stepwise ordering

- Re-order nodes ($p < q$)



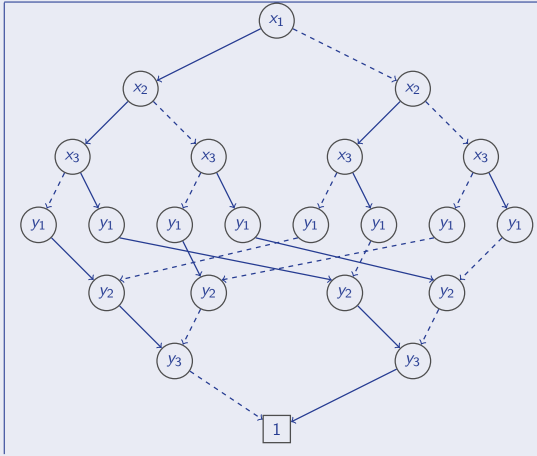
- Eliminate double tests



Variable ordering can make an exponential difference

$(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2) \wedge (x_3 \Leftrightarrow y_3)$ (edges to 0 are suppressed)

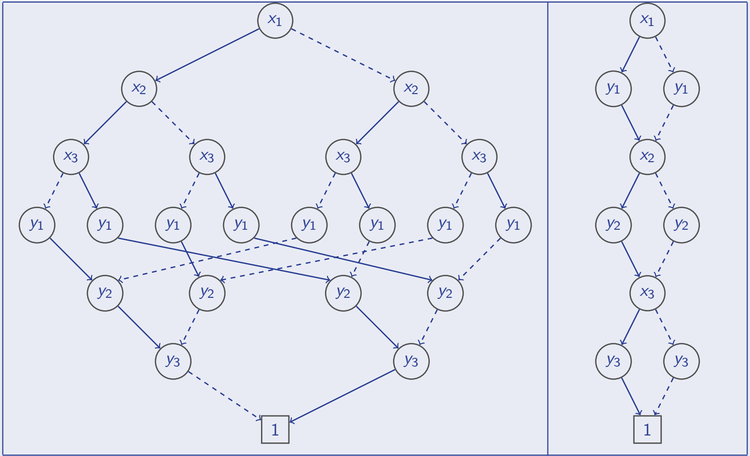
$x_1 < x_2 < x_3 < y_1 < y_2 < y_3$ .... $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$



Variable ordering can make an exponential difference

$(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2) \wedge (x_3 \Leftrightarrow y_3)$ (edges to 0 are suppressed)

$x_1 < x_2 < x_3 < y_1 < y_2 < y_3$ .... $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$



$3.2^n - 2$ nodes | $3.n + 1$ nodes

Theoretical Results

Existence and Uniqueness

- For a fixed variable ordering $(\mathcal{X}, <)$,
- every Boolean function can be represented,
- by a canonical (unique up to isomorphism) OBDD

Theoretical Results

Existence and Uniqueness

- For a fixed variable ordering $(\mathcal{X}, <)$,
- every Boolean function can be represented,
- by a canonical (unique up to isomorphism) OBDD

Ordering

- The chosen ordering has a huge impact on the OBDD size
- Finding the optimal ordering is NP-hard
- Some functions only admit exponentially large OBDDs
  - E.g.: multiplication $P(\vec{x}, \vec{y}, \vec{z})$ such that

$$(x_1 \dots x_n) * (y_1 \dots y_n) = (z_1 \dots z_{2n})$$

    needs $\mathcal{O}(2^n)$ OBDD nodes, whatever ordering is chosen
- In practice, many functions have small OBDD representations
- One distinguishes static and dynamic variable reordering

Table of Contents

OBDD packages

Regard OBDD as abstract datatype

- Manipulation of OBDDs through pointers / objects
- Basic constructors ensure invariant “Reduced & Ordered”
- Operations on OBDDs implement logical connectives:

OBDD packages

Regard OBDD as abstract datatype

- Manipulation of OBDDs through pointers / objects
- Basic constructors ensure invariant "Reduced & Ordered"
- Operations on OBDDs implement logical connectives:

Illustration (5 $<$ 100 functions in C-interface of BuDDy)

```
BDD   bdd_high (BDD r)
BDD   bdd_not (BDD r)
BDD   bdd_apply (BDD l, BDD r, int op)
BDD   bdd_exist (BDD r, BDD var)
BDD   bdd_relprod (BDD l, BDD r, BDD var)
```

OBDD packages

Regard OBDD as abstract datatype

- Manipulation of OBDDs through pointers / objects
- Basic constructors ensure invariant “Reduced & Ordered”
- Operations on OBDDs implement logical connectives:

Illustration (5 < 100 functions in C-interface of BuDDy)

```
BDD    bdd_high (BDD r)
BDD    bdd_not (BDD r)
BDD    bdd_apply (BDD l, BDD r, int op)
BDD    bdd_exist (BDD r, BDD var)
BDD    bdd_relprod (BDD l, BDD r, BDD var)
```

Implementation (we will work with Sylvan, Tom van Dijk)

- Data structures (unique table, operation caches)
- Operations are based on a generic Apply-function

Data structure: Unique Table

Keep maximal sharing and avoid redundant tests

- This is a hash table, to ensure unicity of all BDD nodes
- It assigns a unique number to each triple: $N \leftrightarrow \langle var, N_L, N_H \rangle$
- One can lookup $var(N)$, $low(N)$, $high(N)$ in $\mathcal{O}(1)$ time.

Data structure: Unique Table

Keep maximal sharing and avoid redundant tests

- This is a hash table, to ensure unicity of all BDD nodes
- It assigns a unique number to each triple: $N \leftrightarrow \langle var, N_L, N_H \rangle$
- One can lookup $var(N)$, $low(N)$, $high(N)$ in $\mathcal{O}(1)$ time.

MakeNode$(x, N_L, N_H) = N$ (create new nodes)

Require: variable x, nodes N_L, N_H
Ensure: a unique node N denoting $(\neg x \wedge N_L) \vee (x \wedge N_H)$

if $N_L = N_H$ **then**
    $N := N_L$
else if $\langle x, N_L, N_r \rangle$ is in the unique table **then**
    $N :=$ lookup(x, N_L, N_H)
else
    $N :=$ insert_new_entry(x, N_L, N_H) in the unique table
end if

Naive function for conjunction: $\wedge$

$\text{ApplyAnd}(N_1, N_2) = N$

Require: BDD nodes N_1, N_2
Ensure: BDD node N representing $N_1 \wedge N_2$
1: **if** $N_1 = 0$, $N_1 = 1$, $N_2 = 0$, or $N_2 = 1$ **then**
2: $N := 0$, N_2, 0, N_1, respectively
3: **else**

Naive function for conjunction: $\wedge$

ApplyAnd$(N_1, N_2) = N$

Require: BDD nodes N_1, N_2

Ensure: BDD node N representing $N_1 \wedge N_2$

if $N_1 = 0$, $N_1 = 1$, $N_2 = 0$, or $N_2 = 1$ **then**

  $N := 0$, N_2, 0, N_1, respectively

else

  $x_1, l_1, r_1 := var(N_1), low(N_1), high(N_1)$

  $x_2, l_2, r_2 := var(N_2), low(N_2), high(N_2)$

  if $x_1 = x_2$ **then**

    $N :=$ MakeNode(x_1,ApplyAnd(l_1, l_2),ApplyAnd(r_1, r_2))

  else if $x_1 < x_2$ **then**

    $N :=$ MakeNode(x_1,ApplyAnd(l_1, N_2),ApplyAnd(r_1, N_2))

  else if $x_1 > x_2$ **then**

    $N :=$ MakeNode(x_2,ApplyAnd(N_1, l_2),ApplyAnd(N_1, r_2))

  end if

end if

Problem with naive recursion

Naive Complexity

- Consider a BDD with n nodes, but a lot of sharing
- The BDD can have $O(2^n)$ different paths (!)
- Hence the naive APPLY takes $O(2^n)$ recursive calls

Problem with naive recursion

Naive Complexity

- Consider a BDD with n nodes, but a lot of sharing
- The BDD can have $O(2^n)$ different paths (!)
- Hence the naive APPLY takes $O(2^n)$ recursive calls

Solution: Dynamic Programming

- Store all intermediate results in an operation cache
  - first check if the result is already in the cache
  - if not, compute it and put the result in the cache
- this is a well-known technique: Fibonacci sequence

Problem with naive recursion

Naive Complexity

- Consider a BDD with n nodes, but a lot of sharing
- The BDD can have $O(2^n)$ different paths (!)
- Hence the naive APPLY takes $O(2^n)$ recursive calls

Solution: Dynamic Programming

- Store all intermediate results in an operation cache
  - first check if the result is already in the cache
  - if not, compute it and put the result in the cache
- this is a well-known technique: Fibonacci sequence

Ultimate Complexity

- Given OBDDs A with m nodes and B with n nodes,
- There can be at most $m \cdot n$ pairs of nodes from A and B.
- With dynamic programming, APPLY takes $\mathcal{O}(m \cdot n)$ steps.

Data structure: Operation Cache

Apply$(op, N_1, N_2) = N$ .... (recursive cases only)

if (op, N_1, N_2) is in the operation cache **then**
    $N :=$ lookup(op, N_1, N_2)
else
    $x_1, l_1, r_1 := var(N_1), low(N_1), high(N_1)$
    $x_2, l_2, r_2 := var(N_2), low(N_2), high(N_2)$
    if $x_1 = x_2$ **then**
        $N :=$ MakeNode(x_1,Apply(op,l_1, l_2),Apply(op,r_1, r_2))
    else if $x_1 < x_2$ **then**
        $N :=$ MakeNode(x_1,Apply(op,l_1, N_2),Apply(op,r_1, N_2))
    else if $x_1 > x_2$ **then**
        $N :=$ MakeNode(x_2,Apply(op,N_1, l_2),Apply(op,N_1, r_2))
    end if
    add $(op, N_1, N_2) \mapsto N$ to the operation cache
end if

Why is your BDD package better than mine?

Black magic

- Variable ordering: sometimes even dynamically reordered
- Garbage collection on the unique table
- Operation cache replacement strategy
- And even: effect on L2 cache versus main memory

Why is your BDD package better than mine?

Black magic

- Variable ordering: sometimes even dynamically reordered
- Garbage collection on the unique table
- Operation cache replacement strategy
- And even: effect on L2 cache versus main memory

And what about the user?

- Performance depends on how the BDD package is used
- Start with a good initial variable ordering
- Think about the order of applying operations

Table of Contents

Kripke structures

Definition

A Kripke structure is a tuple (S, S_0, R, AP, L), where

- S is a set of states
- $S_0 \subseteq S$ is set of initial states
- $R \subseteq S \times S$ is a total transition relation on S
  - $\forall s \in S. \exists t \in S. R(s, t)$
- AP is a set of atomic proposition labels
- $L : S \to \mathcal{P}(AP)$ assigns to each state a set of labels

Kripke structures

Definition

A Kripke structure is a tuple (S, S_0, R, AP, L), where

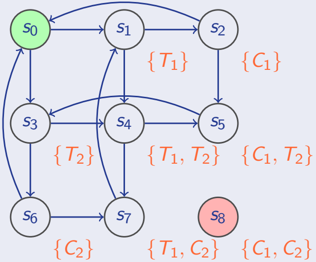
- S is a set of states
- $S_0 \subseteq S$ is set of initial states
- $R \subseteq S \times S$ is a total transition relation on S
  - $\forall s \in S. \exists t \in S. R(s, t)$
- AP is a set of atomic proposition labels
- $L : S \to \mathcal{P}(AP)$ assigns to each state a set of labels

Kripke structures versus transition systems

- Note that here we put the labels on the states, while in process algebra, we put labels on the edges (labeled transition systems)

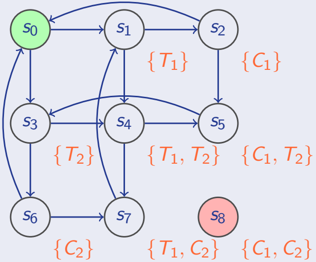
Example of Kripke Structure

Mutual Exclusion / Critical Section



Example of Kripke Structure

Mutual Exclusion / Critical Section



Parts of the Kripke Structure tuple:

- States $S = \{s_0, \ldots, s_8\}$; Initial states $S_0 = \{s_0\}$.
- $R = \{(s_0, s_1), (s_1, s_2), \ldots\}$
- $AP = \{T_1, C_1, T_2, C_2\}$ (trying, critical)
- $L(s_0) = \varnothing$; $L(s_4) = \{T_1, T_2\}$; $L(s_7) = \{T_1, C_2\}, \ldots$

Boolean Encoding of Kripke Structure: states

Encoding in Booleans (=bits)

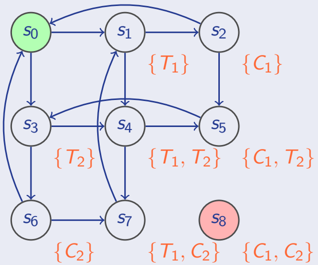
- Virtually everything can be encoded in bits (as you know)
- We would like to preserve structure as much as possible
- Here we choose the atomic propositions to encode states (this is often done, but not always possible/necessary)

Boolean Encoding of Kripke Structure: states

Encoding in Booleans (=bits)

- Virtually everything can be encoded in bits (as you know)
- We would like to preserve structure as much as possible
- Here we choose the atomic propositions to encode states (this is often done, but not always possible/necessary)

Encoding of this example: use variables $\{T_1, T_2, C_1, C_2\}$

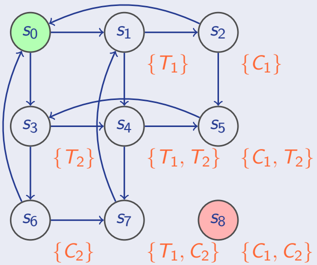


Boolean Encoding of Kripke Structure: states

Encoding in Booleans (=bits)

- Virtually everything can be encoded in bits (as you know)
- We would like to preserve structure as much as possible
- Here we choose the atomic propositions to encode states (this is often done, but not always possible/necessary)

Encoding of this example: use variables $\{T_1, T_2, C_1, C_2\}$



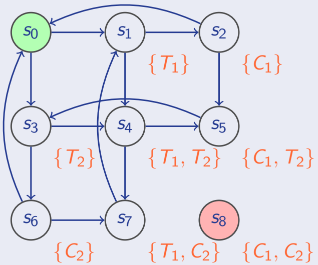
- States correspond to formulas:
  $s_0 = \neg T_1 \wedge \neg T_2 \wedge \neg C_1 \wedge \neg C_2$
  $s_4 = T_1 \wedge T_2 \wedge \neg C_1 \wedge \neg C_2$

Boolean Encoding of Kripke Structure: states

Encoding in Booleans (=bits)

- Virtually everything can be encoded in bits (as you know)
- We would like to preserve structure as much as possible
- Here we choose the atomic propositions to encode states (this is often done, but not always possible/necessary)

Encoding of this example: use variables $\{T_1, T_2, C_1, C_2\}$



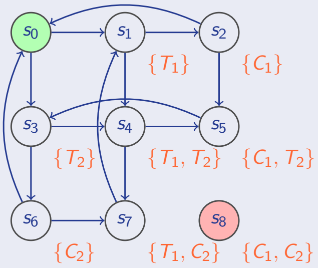
- States correspond to formulas:
  $s_0 = \neg T_1 \wedge \neg T_2 \wedge \neg C_1 \wedge \neg C_2$
  $s_4 = T_1 \wedge T_2 \wedge \neg C_1 \wedge \neg C_2$
- Set of states are also formulas:
  $\{s_1, s_4, s_7\} = T_1$
  $\{s_6, s_7, s_8\} = C_2$
- Set of reachable states?

Boolean Encoding of Kripke Structure: states

Encoding in Booleans (=bits)

- Virtually everything can be encoded in bits (as you know)
- We would like to preserve structure as much as possible
- Here we choose the atomic propositions to encode states (this is often done, but not always possible/necessary)

Encoding of this example: use variables $\{T_1, T_2, C_1, C_2\}$



- States correspond to formulas:
  $s_0 = \neg T_1 \wedge \neg T_2 \wedge \neg C_1 \wedge \neg C_2$
  $s_4 = T_1 \wedge T_2 \wedge \neg C_1 \wedge \neg C_2$
- Set of states are also formulas:
  $\{s_1, s_4, s_7\} = T_1$
  $\{s_6, s_7, s_8\} = C_2$
- Set of reachable states?
  $\neg(C_1 \wedge C_2)$ ... (details?)

Boolean Encoding of Kripke Structure: transitions

Encoding of States and Transitions

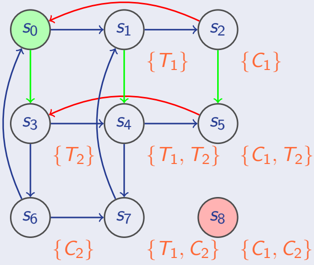
- We have encoded sets of states as formulas $P(T_1, T_2, C_1, C_2)$.
- Transitions relate (T_1, T_2, C_1, C_2) and (T_1', T_2', C_1', C_2')
- Encode transitions as formulas: $Q(T_1, T_1', T_2, T_2', C_1, C_1', C_2, C_2')$.

Boolean Encoding of Kripke Structure: transitions

Encoding of States and Transitions

- We have encoded sets of states as formulas $P(T_1, T_2, C_1, C_2)$.
- Transitions relate (T_1, T_2, C_1, C_2) and (T_1', T_2', C_1', C_2')
- Encode transitions as formulas: $Q(T_1, T_1', T_2, T_2', C_1, C_1', C_2, C_2')$.

Encoding of transitions:



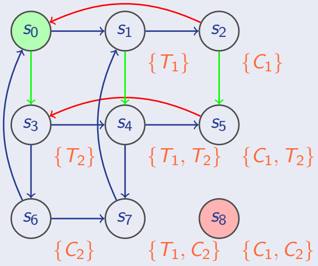
- Represent transitions by formulas:
  - Green: $\neg T_2 \wedge \neg C_2 \wedge T_2' \wedge \neg C_2'$
  - Red: $C_1 \wedge \neg C_2 \wedge \neg C_1' \wedge \neg T_1'$

Boolean Encoding of Kripke Structure: transitions

Encoding of States and Transitions

- We have encoded sets of states as formulas $P(T_1, T_2, C_1, C_2)$.
- Transitions relate (T_1, T_2, C_1, C_2) and (T_1', T_2', C_1', C_2')
- Encode transitions as formulas:
  $Q(T_1, T_1', T_2, T_2', C_1, C_1', C_2, C_2')$.

Encoding of transitions:



- Represent transitions by formulas:
  - Green: $\neg T_2 \wedge \neg C_2 \wedge T_2' \wedge \neg C_2'$
  - Red: $C_1 \wedge \neg C_2 \wedge \neg C_1' \wedge \neg T_1'$
- This is not quite true: also ensure other variables don't change:
  - Green: $T_1 = T_1' \wedge C_1 = C_1'$.
  - Red: $T_2 = T_2' \wedge C_2 = C_2'$.

Table of Contents

Next-state by Relational Product

Relational Product: the problem

- Given: some set of states S, and a relation R
- Represented by: $P(\vec{x})$ and $Q(\vec{x}, \vec{x'})$.
- Required: the R-successors of S, i.e. $\{t \mid \exists s \in S . s\, R\, t\}$

Next-state by Relational Product

Relational Product: the problem

- Given: some set of states S, and a relation R
- Represented by: $P(\vec{x})$ and $Q(\vec{x}, \vec{x'})$.
- Required: the R-successors of S, i.e. $\{t \mid \exists s \in S.\, s\, R\, t\}$

Relational product: the solution

- Step 1: Simply take the conjunction: $P(\vec{x}) \wedge Q(\vec{x}, \vec{x'})$
- Step 2: Abstract previous states: $\exists \vec{x}.\, P(\vec{x}) \wedge Q(\vec{x}, \vec{x'})$
- Step 3: Rename back to original state variable names: $\left(\exists \vec{x}.\, P(\vec{x}) \wedge Q(\vec{x}, \vec{x'})\right)[\vec{x}/\vec{x'}]$

Next-state by Relational Product

Relational Product: the problem

- Given: some set of states S, and a relation R
- Represented by: $P(\vec{x})$ and $Q(\vec{x}, \vec{x'})$.
- Required: the R-successors of S, i.e. $\{t \mid \exists s \in S.\, s\, R\, t\}$

Relational product: the solution

- Step 1: Simply take the conjunction: $P(\vec{x}) \wedge Q(\vec{x}, \vec{x'})$
- Step 2: Abstract previous states: $\exists \vec{x}.\, P(\vec{x}) \wedge Q(\vec{x}, \vec{x'})$
- Step 3: Rename back to original state variable names: $\left(\exists \vec{x}.\, P(\vec{x}) \wedge Q(\vec{x}, \vec{x'})\right)[\vec{x}/\vec{x'}]$

What about existential quantification?

- $\exists x_i.\, P(\vec{x})$ is simply: $P[x_i := 0] \vee P[x_i := 1]$
- Note: size can double, so $\exists \vec{x}.P$ can be exponentially big!

We are finished!

Recall Set-BFS (variant 1)

$Vis := Cur := \{\mathrm{init}\}$
while $Cur \neq \emptyset$ **do**
  check $Cur \cap \mathrm{Error} = \emptyset$
  $Cur := \mathrm{Next}(Cur, \rightarrow) \setminus Vis$
  $Vis := Vis \cup Cur$
end while

Recall Set-BFS (variant 2)

$V_{\mathrm{old}} := \emptyset$
$V_{\mathrm{new}} := \{\mathrm{init}\}$
while $V_{\mathrm{old}} \neq V_{\mathrm{new}}$ **do**
  $V_{\mathrm{old}} := V_{\mathrm{new}}$
  $V_{\mathrm{new}} := V_{\mathrm{old}} \cup \mathrm{Next}(V_{\mathrm{old}}, \rightarrow)$
end while

We are finished!

Recall Set-BFS (variant 1)

$Vis := Cur := \{\text{init}\}$
while $Cur \neq \varnothing$ **do**
  check $Cur \cap \text{Error} = \varnothing$
  $Cur := \text{Next}(Cur, \rightarrow) \setminus Vis$
  $Vis := Vis \cup Cur$
end while

Recall Set-BFS (variant 2)

$V_{\text{old}} := \varnothing$
$V_{\text{new}} := \{\text{init}\}$
while $V_{\text{old}} \neq V_{\text{new}}$ **do**
  $V_{\text{old}} := V_{\text{new}}$
  $V_{\text{new}} := V_{\text{old}} \cup \text{Next}(V_{\text{old}}, \rightarrow)$
end while

Implementation with BDDs

- Vis, Cur: Binary Decision Diagrams (BDDs)
- $\cap$: BDDapplyAnd
- $\cup$: BDDapplyOr
- $Next$: BDDRelProd
- $\neq$: pointer comparison (unique representation!)

Table of Contents

Partitioning of the next-state function

Realistic systems are composed naturally

- Each component uses only a subvector of the state variables
- Each component i has its own (simple) next-state relation R_i

Partitioning of the next-state function

Realistic systems are composed naturally

- Each component uses only a subvector of the state variables
- Each component i has its own (simple) next-state relation R_i

Synchronous systems: conjunctive partitioning

- In hardware: all components do a step at clock ticks
- So the relation of the system is: $R_1 \wedge R_2 \wedge \cdots \wedge R_n$.

Partitioning of the next-state function

Realistic systems are composed naturally

- Each component uses only a subvector of the state variables
- Each component i has its own (simple) next-state relation R_i

Synchronous systems: conjunctive partitioning

- In hardware: all components do a step at clock ticks
- So the relation of the system is: $R_1 \wedge R_2 \wedge \cdots \wedge R_n$.

A-synchronous systems: disjunctive partitioning

- In parallel software: transitions are interleaved, non-determinism
- So the relation of the system is: $R_1 \vee R_2 \vee \cdots \vee R_n$.
- (In practice also: "the other variables are unchanged")

Four tricks to make symbolic methods more efficient

Disjunctive Partitioning

- Handle all subtransitions R_i separately
- Never compute the expensive $R = R_1 \vee \cdots \vee R_n$

Four tricks to make symbolic methods more efficient

Disjunctive Partitioning

- Handle all subtransitions R_i separately
- Never compute the expensive $R = R_1 \vee \cdots \vee R_n$

Chaining

- Apply R_2 on the result of applying R_1, etc.
- Basically, compute $(R_2 \circ R_1)^*(S)$ instead of $(R_1 \cup R_n)^*(S)$

Four tricks to make symbolic methods more efficient

Disjunctive Partitioning

- Handle all subtransitions R_i separately
- Never compute the expensive $R = R_1 \vee \cdots \vee R_n$

Chaining

- Apply R_2 on the result of applying R_1, etc.
- Basically, compute $(R_2 \circ R_1)^*(S)$ instead of $(R_1 \cup R_n)^*(S)$

Relational Product

- Interweave the EXIST and APPLY operations
- Abstract variables as soon as they are introduced by APPLY

Four tricks to make symbolic methods more efficient

Disjunctive Partitioning

- Handle all subtransitions R_i separately
- Never compute the expensive $R = R_1 \vee \cdots \vee R_n$

Chaining

- Apply R_2 on the result of applying R_1, etc.
- Basically, compute $(R_2 \circ R_1)^*(S)$ instead of $(R_1 \cup R_n)^*(S)$

Relational Product

- Interweave the EXIST and APPLY operations
- Abstract variables as soon as they are introduced by APPLY

Variable Ordering

- Keep variables from same component together; also x and x'
- Sophisticated: dynamically reorder variables during computation

Symbolic Reachability algorithm, revisited

Reachable(I, N, R_i) = V_{new}

Require: I, BDD representing initial states
Require: R_i, BDDs, representing subtransitions
Ensure: V_{new} BDD representing the reachable states from I by R_i

```
  V_old := BDDempty
  V_new := I
  while not BDDequal(V_old, V_new) do
    V_old := V_new
    for i = 1 to N do
      V_new := BDDapply(∨, V_new, BDDrelprod(V_new, R_i))
    end for
  end while
```

Symbolic Reachability algorithm, revisited

Reachable(I, N, R_i) = V_{new}

Require: I, BDD representing initial states
Require: R_i, BDDs, representing subtransitions
Ensure: V_{new} BDD representing the reachable states from I by R_i

V_{old} := BDDempty
V_{new} := I
while not BDDequal($V_{\text{old}}, V_{\text{new}}$) **do**
  V_{old} := V_{new}
  for $i = 1$ to N **do**
    $V_{\text{new}} := BDDapply(\vee, V_{\text{new}}, BDDrelprod(V_{\text{new}}, R_i))$
  end for
end while