19. Symbolic Model Checking

Computer–Aided Verification

Dave Parker

University of Birmingham

2017/18

Rest of the module

- Lectures
  - today: symbolic model checking
  - Thursday: probabilistic model checking
  - no lectures next week

- Assignments & exercises
  - Assignment 3 – solutions on Canvas (and Q2ii remarked)
  - Assignment 4 (SPIN) – out now, due Thursday of week 11
  - Assignment 5 (extended only) – due Thursday
  - non-assessed exercise (bounded model checking) online

Module syllabus

- Modelling sequential and parallel systems
  - labelled transitions systems, parallel composition
- Temporal logic
  - LTL, CTL and CTL*, etc.
- Model checking
  - CTL model checking algorithms
  - automata-theoretic model checking (LTL)
- Verification tools: SPIN
- Advanced verification techniques
  - bounded model checking via propositional satisfiability
  - symbolic model checking
  - probabilistic model checking

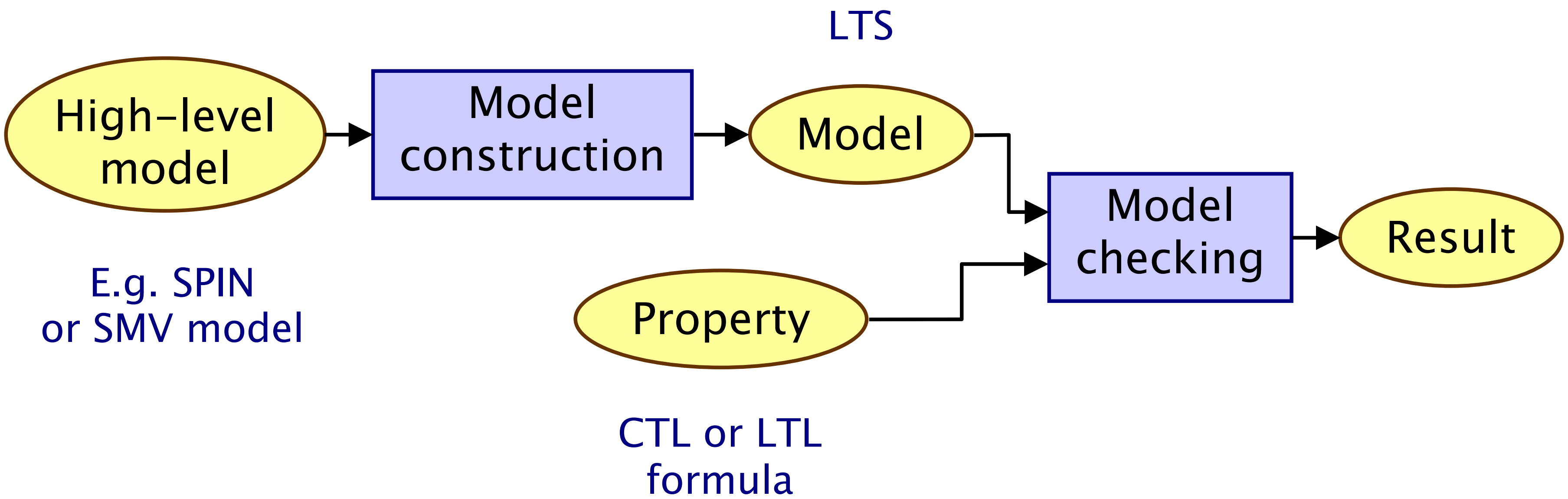
Overview

- Last time
  - bounded model checking via SAT (or SMT)
  - “symbolic” encoding of model checking problem
  - targets falsification up to a finite number of unwindings
  - can be made complete, e.g. with k-induction

- This lecture
  - symbolic model checking
  - binary decision diagrams (BDDs)
  - exploits regularity to improve scalability of model checking
  - i.e. targets state space explosion problem
  - well suited to verification (as opposed to falsification)
  - applicable much more widely

Model checking implementation

- Overview of the model checking process
  - two phases: model construction, model checking
  - several different logics, multiple algorithms
  - but… they have various aspects/operations in common
    - basic set operations, reachability, strongly connected components, …
    - manipulation of transition relation and state sets



Explicit vs. symbolic data structures

- Symbolic data structures
  - usually based on binary decision diagrams (BDDs) or variants
  - avoid explicit enumeration of data by exploiting regularity
  - potentially very compact/efficient storage (but not always)

- Sets of states:
  - explicit: bit vectors, hashing
  - symbolic: BDDs

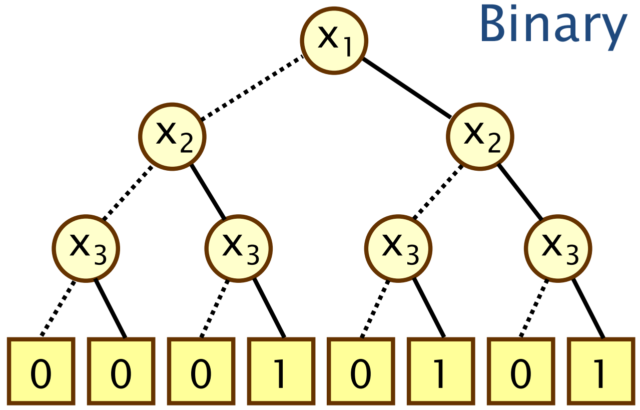
- Transition relations:
  - explicit: sparse adjacency matrix
  - symbolic: BDDs

Representations of Boolean formulas

- Propositional formula: $f = (x_1 \vee x_2) \wedge x_3$

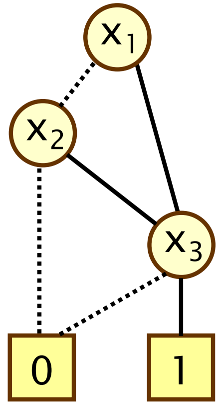
Truth table

x_1	x_2	x_3	f
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1



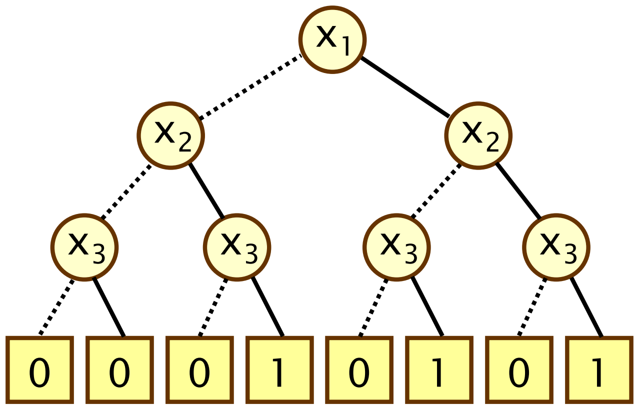
Binary decision tree

Binary decision diagram



Binary decision trees

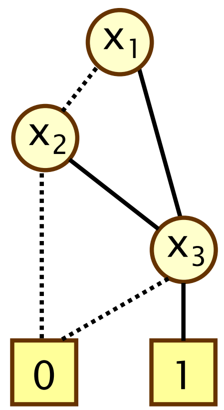
- Graphical representation of Boolean functions
  - $f(x_1,\ldots,x_n) : \{0,1\}^n \rightarrow \{0,1\}$
- Binary tree with two types of nodes
- Non-terminal nodes
  - labelled with a Boolean variable x_i
  - two children: 1 (“then”, solid line) and 0 (“else”, dotted line)
- Terminal nodes (or “leaf” nodes)
  - labelled with 0 or 1
- To read the value of $f(x_1,\ldots,x_n)$
  - start at root (top) node
  - take “then” edge if $x_i=1$
  - take “else” edge if $x_i=0$
  - result given by leaf node



Binary decision diagrams

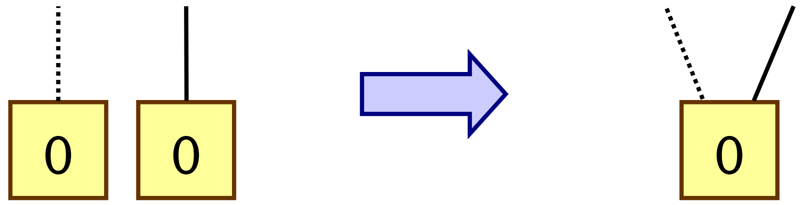
- Binary decision diagrams (BDDs)
  - based on binary decision trees, but reduced and ordered
  - sometimes called reduced ordered BDDs (ROBDDs)
  - actually directed acyclic graphs (DAGs), not trees
  - compact, canonical representation for Boolean functions

- Variable ordering
  - a BDD assumes a fixed total ordering over its set of Boolean variables
  - e.g. $x_1<x_2<x_3$
  - along any path through the BDD, variables appear at most once each and always in the correct order

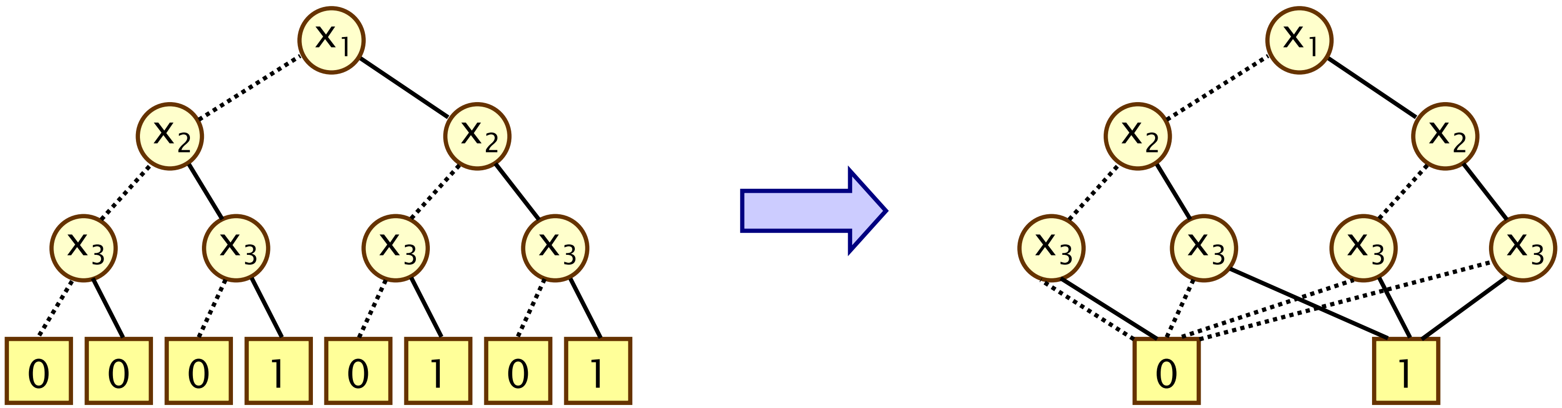


BDD reduction rule 1

- Rule 1: Merge identical terminal nodes

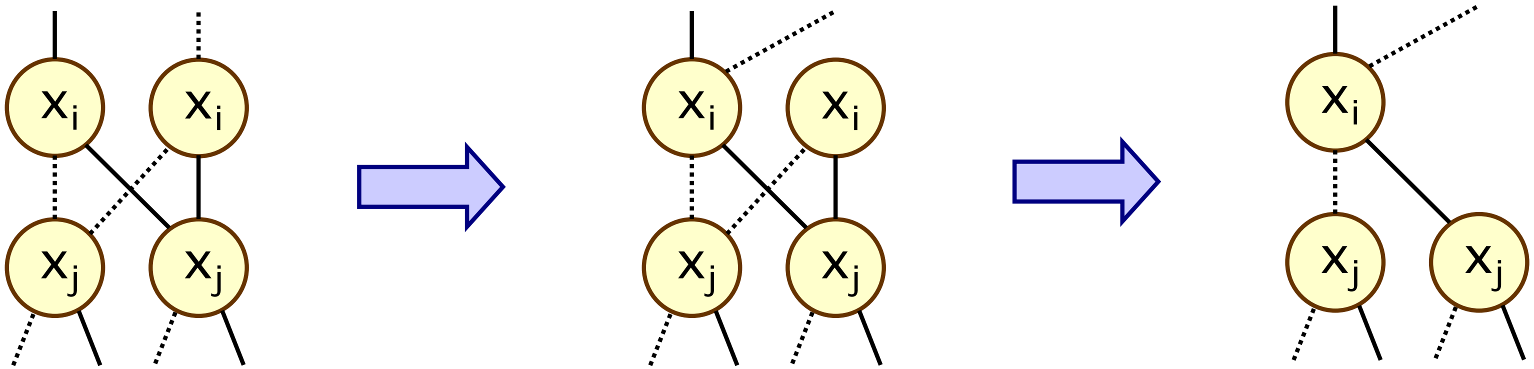


- Example:

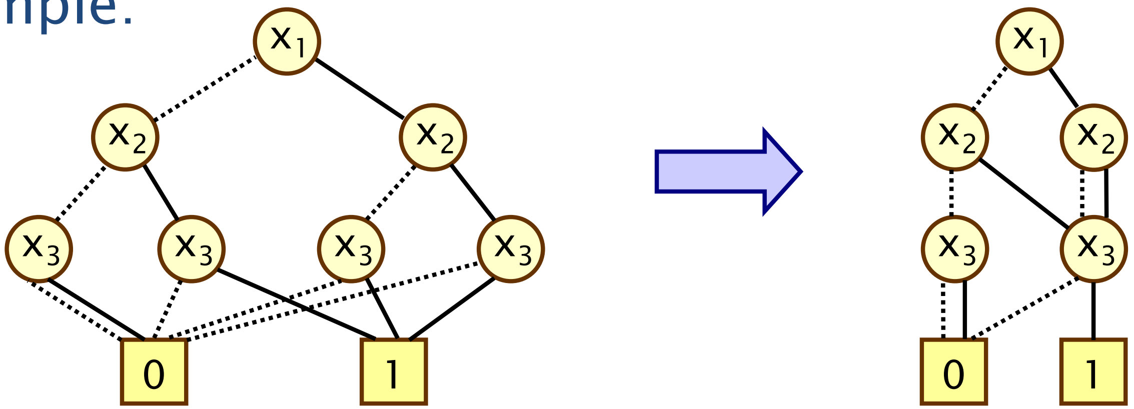


BDD reduction rule 2

- Rule 2: Merge isomorphic nodes, redirect incoming nodes

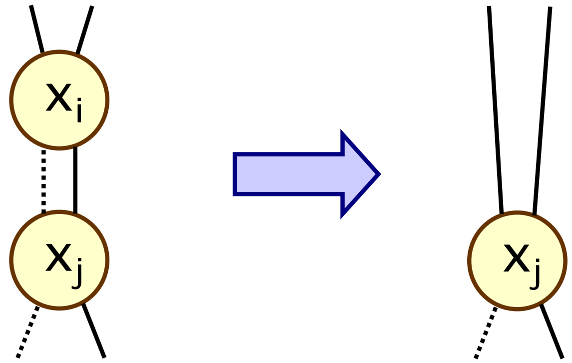


- Example:

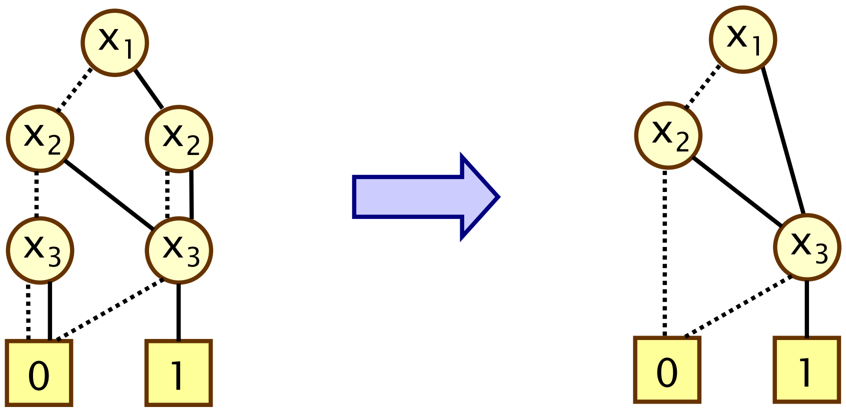


BDD reduction rule 3

- Rule 3: Remove redundant nodes (with identical children)

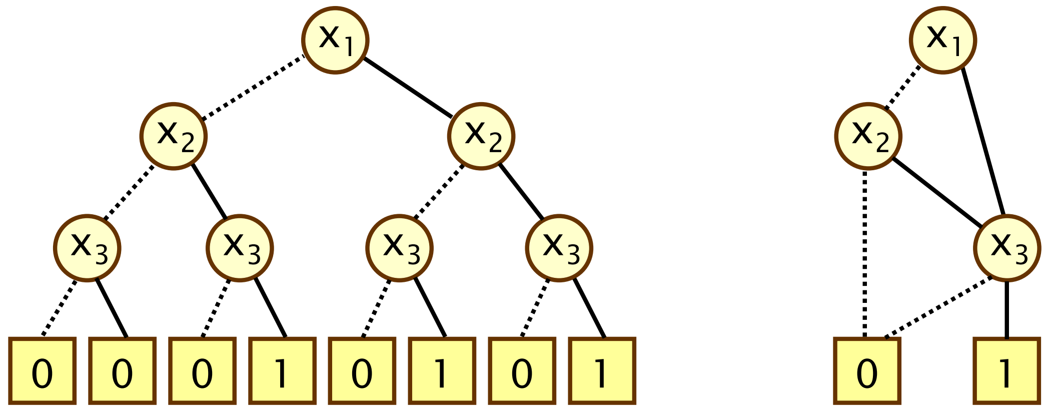


- Example:



Canonicity

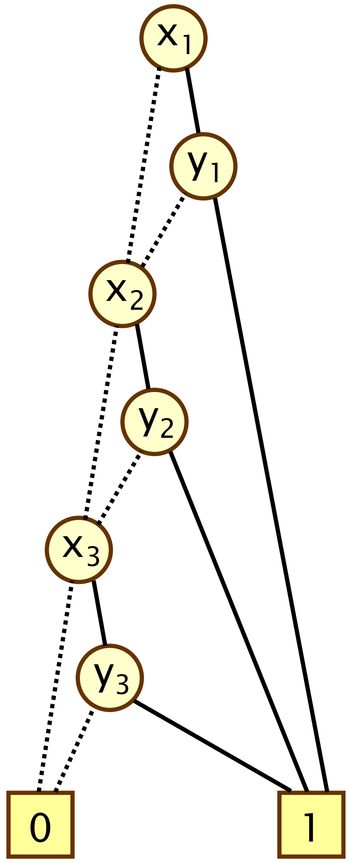
- BDDs are a canonical representation for Boolean functions
  - two Boolean functions are equivalent if and only if the BDDs which represent them are isomorphic
  - uniqueness relies on: reduced BDDs, fixed variable ordered



- Important implications for implementation efficiency
  - can be tested in linear (or even constant) time

BDD variable ordering

- BDD size can be very sensitive to the variable ordering
  - example: $f = (x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$
  - two orderings:
    - $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$
    - $x_1 < x_2 < x_3 < y_1 < y_2 < y_3$
  - which is better?

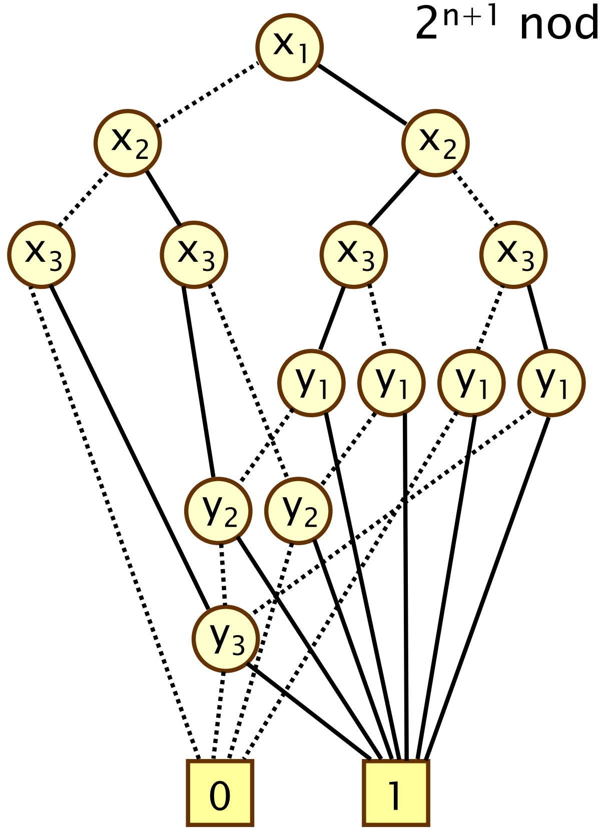


$x_1 < y_1 < x_2 < y_2 < x_3 < y_3$

2n+2 nodes

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

2^{n+1} nodes



BDDs to represent sets of states

- Consider a state space S and some subset $S' \subseteq S$
- We can represent S' by its characteristic function $\chi_{S'}$
  - $\chi_{S'} : S \rightarrow \{0,1\}$ where $\chi_{S'}(s) = 1$ if and only if $s \in S'$
- Assume we have an encoding of S into n Boolean variables
  - this is always possible for a finite set S
  - e.g. enumerate the elements of S and use a binary encoding
  - (note: there may be more efficient encodings though)
- So $\chi_{S'}$ can be seen as a function $\chi_{S'}(x_1,\ldots x_n) : \{0,1\}^n \rightarrow \{0,1\}$
  - which is simply a Boolean function
  - which can therefore be represented as a BDD

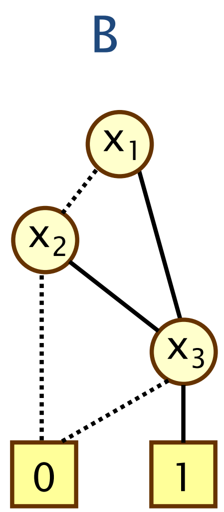
BDD and sets of states – Example

- State space S: {0, 1, 2, 3, 4, 5, 6, 7}
- Encoding of S: {000, 001, 010, 011, 100, 101, 110, 111}
- Subset S’ ⊆ S: {3, 5, 7} → {011, 101, 111}

Truth table:

x_1	x_2	x_3	f_B
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

BDD:

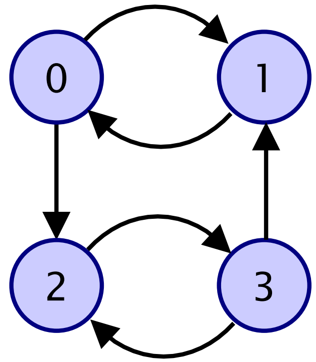


BDDs and transition relations

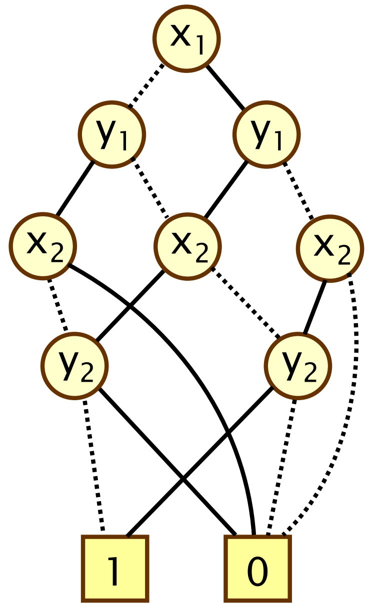
- Transition relations can also be represented by their characteristic function, but over pairs of states
  - relation: $R \subseteq S \times S$
  - characteristic function: $\chi_R : S \times S \to \{0,1\}$
- For an encoding of state space S into n Boolean variables
  - we have Boolean function $f_R(x_1,\ldots,x_n,y_1,\ldots,y_n) : \{0,1\}^{2n} \to \{0,1\}$
  - which can be represented by a BDD
- Row and column variables
  - for efficiency reasons, we interleave the row variables $x_1,..,x_n$ and column variables $y_1,\ldots,y_n$
  - i.e. we use function $f_R(x_1,y_1,\ldots,x_n,y_n) : \{0,1\}^{2n} \to \{0,1\}$

BDDs and transition relations

- Example:
  - 4 states: 0, 1, 2, 3
  - Encoding: $0 \mapsto 00$, $1 \mapsto 01$, $2 \mapsto 10$, $3 \mapsto 11$



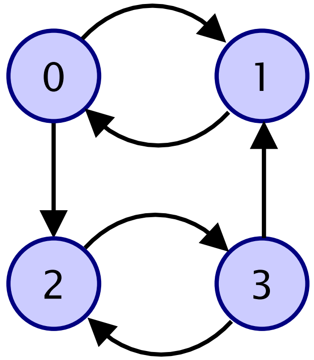
Transition	x_1	x_2	y_1	y_2	$x_1y_1x_2y_2$
(0,1)	0	0	0	1	0001
(0,2)	0	0	1	0	0100
(1,0)	0	1	0	0	0010
(2,3)	1	0	1	1	1101
(3,1)	1	1	0	1	1011
(3,2)	1	1	1	0	1110



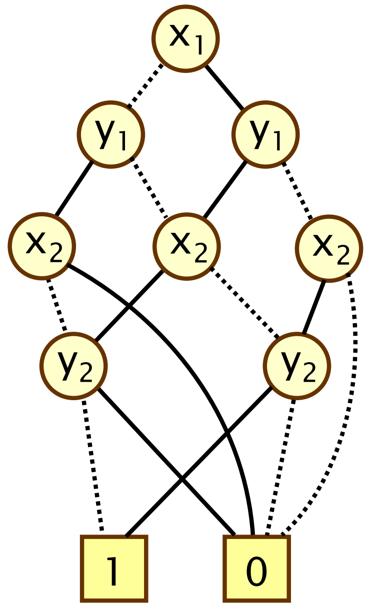
BDDs and transition relations

- We can also think of the transition relation as an adjacency matrix

$$\begin{array}{|cccc|}\hline 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ \hline\end{array}$$

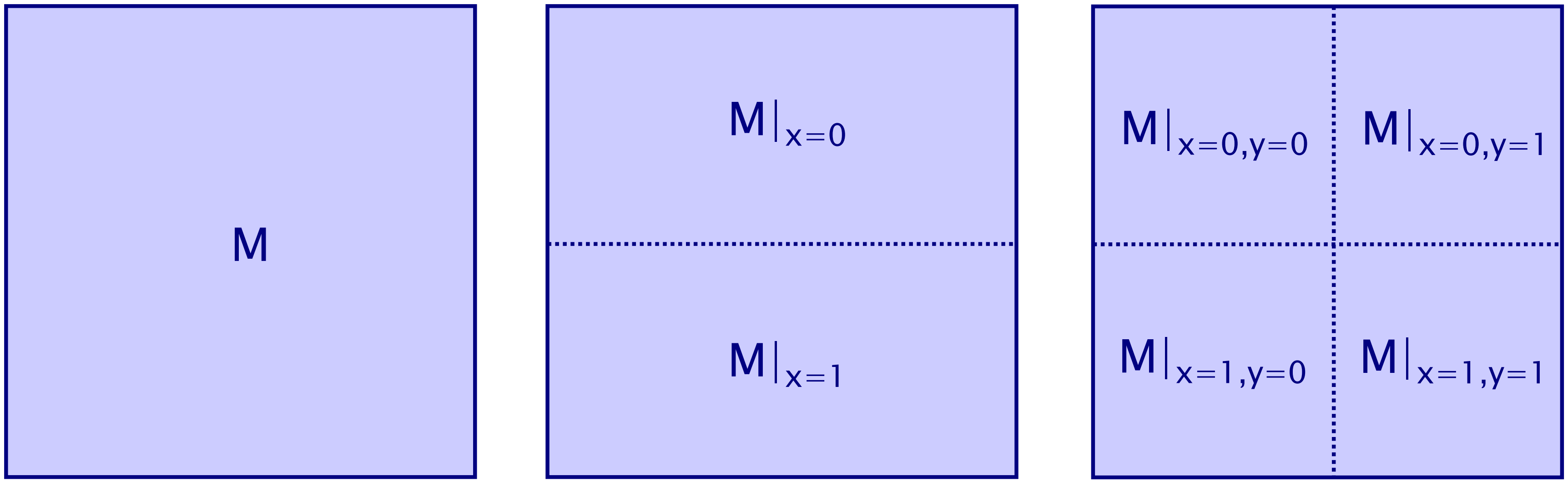


Transition	x_1	x_2	y_1	y_2	$x_1y_1x_2y_2$
(0,1)	0	0	0	1	0001
(0,2)	0	0	1	0	0100
(1,0)	0	1	0	0	0010
(2,3)	1	0	1	1	1101
(3,1)	1	1	0	1	1011
(3,2)	1	1	1	0	1110



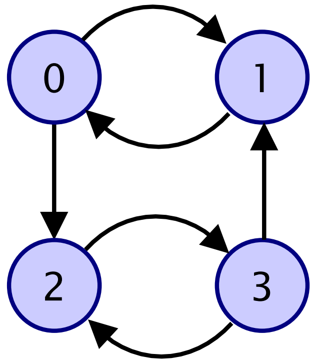
Matrices and BDDs – Recursion

- Descending one level in the BDD (i.e. setting $x_i=b$)
  - splits the matrix represented by the BDD in half
  - row variables (x_i) give horizontal split
  - column variables (y_i) give vertical split

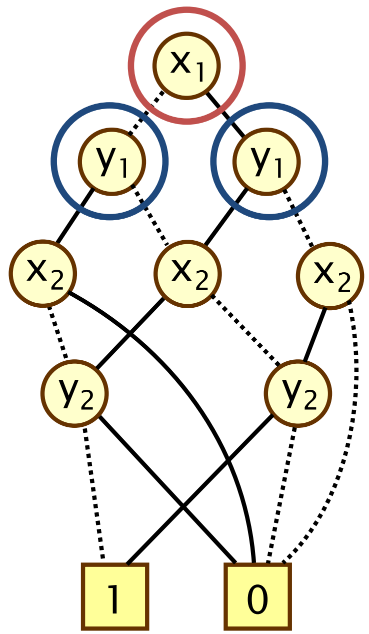


Matrices and BDDs – Recursion

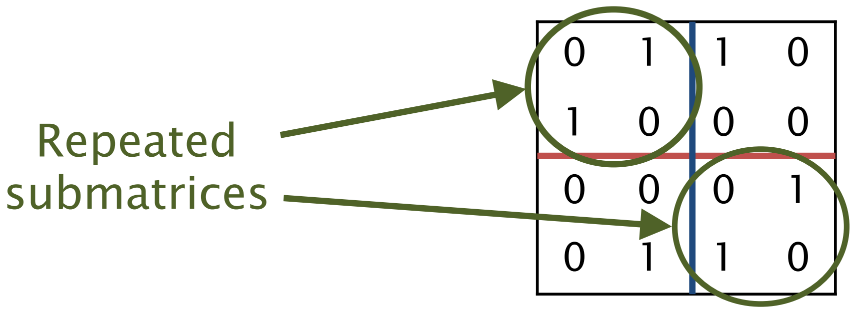
$$\left[\begin{array}{cc|cc} 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{array}\right]$$

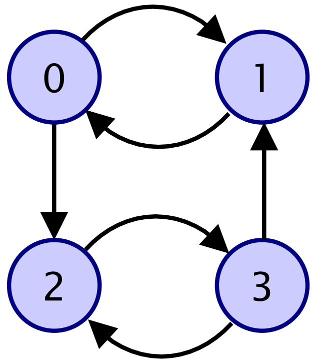


Transition	x_1	x_2	y_1	y_2	$x_1y_1x_2y_2$
(0,1)	0	0	0	1	0001
(0,2)	0	0	1	0	0100
(1,0)	0	1	0	0	0010
(2,3)	1	0	1	1	1101
(3,1)	1	1	0	1	1011
(3,2)	1	1	1	0	1110

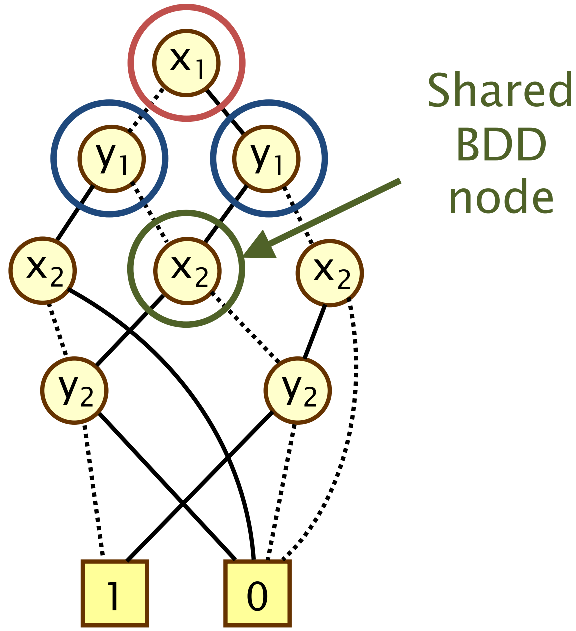


Matrices and BDDs – Regularity





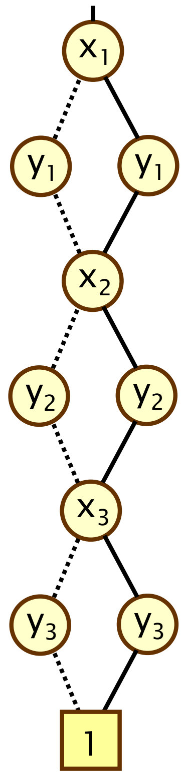
Transition	x_1	x_2	y_1	y_2	$x_1y_1x_2y_2$
(0,1)	0	0	0	1	0001
(0,2)	0	0	1	0	0100
(1,0)	0	1	0	0	0010
(2,3)	1	0	1	1	1101
(3,1)	1	1	0	1	1011
(3,2)	1	1	1	0	1110



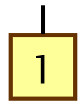
Matrices and BDDs – Compactness

- Some simple matrices (or relations) have extremely compact representations as BDDs
  - e.g. the identify matrix or a constant matrix

$$\begin{array}{|cccccccc|}\hline 1&0&0&0&0&0&0&0\\ 0&1&0&0&0&0&0&0\\ 0&0&1&0&0&0&0&0\\ 0&0&0&1&0&0&0&0\\ 0&0&0&0&1&0&0&0\\ 0&0&0&0&0&1&0&0\\ 0&0&0&0&0&0&1&0\\ 0&0&0&0&0&0&0&1\\ \hline\end{array}$$



$$\begin{array}{|cccccccc|}\hline 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1\\ 1&1&1&1&1&1&1&1\\ \hline\end{array}$$



Flashback: CTL model checking ∃U

- Procedure to compute $Sat(\exists(\phi_1 \ U \ \phi_2))$
  - given $Sat(\phi_1)$ and $Sat(\phi_2)$

- Basic idea: backwards search of the LTS from ϕ_2-states
  - $T_0 := Sat(\phi_2)$
  - $T_i := T_{i-1} \cup \{ s \in Sat(\phi_1) \mid Post(s) \cap T_{i-1} \neq \emptyset \}$
  - until $T_i = T_{i-1}$
  - $Sat(\exists(\phi_1 \ U \ \phi_2)) = T_i$
- (i.e. keep adding predecessors of states in T_{i-1})

- Based on expansion law
  - $\exists(\phi_1 \ U \ \phi_2) \equiv \phi_2 \vee (\phi_1 \wedge \exists\bigcirc \exists(\phi_1 \ U \ \phi_2))$
  - (can be formulated as a fixed-point equation)

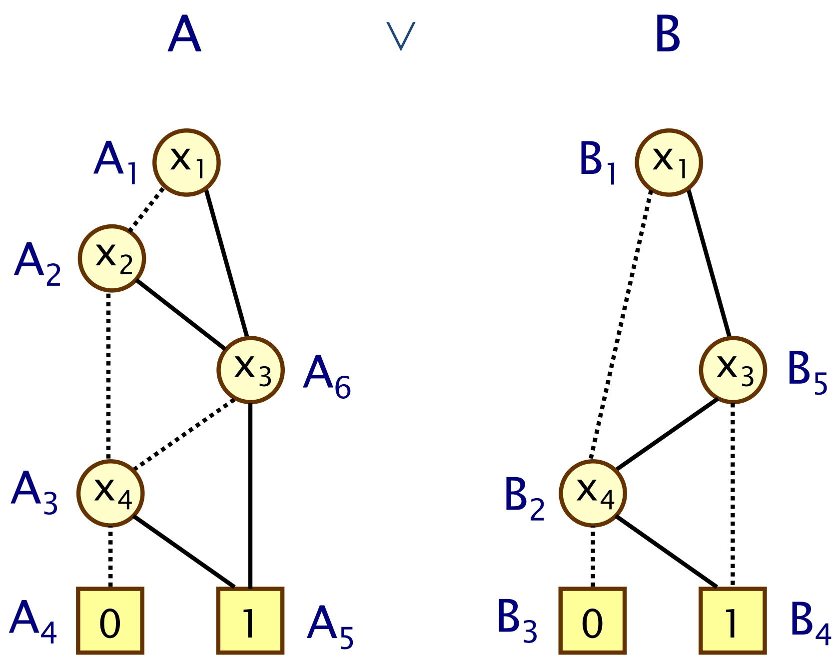
Manipulating BDDs

- Need efficient ways to manipulate Boolean functions
  - while they are represented as BDDs
  - i.e. algorithms which are applied directly to the BDDs
- Basic operations on Boolean functions:
  - negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), etc.
  - can all be applied directly to BDDs
- Key operation on BDDs: Apply(op, A, B)
  - where A and B are BDDs and op is a binary operator over Boolean values, e.g. $\wedge$, $\vee$, etc.
  - Apply(op, A, B) returns the BDD representing function f_A op f_B
  - often just use infix notation, e.g. Apply($\wedge$, A, B) = A $\wedge$ B
  - efficient algorithm: recursive depth-first traversal of A and B
  - complexity (and size of result) is O($|A| \cdot |B|$)
    - where $|C|$ denotes size of BDD C

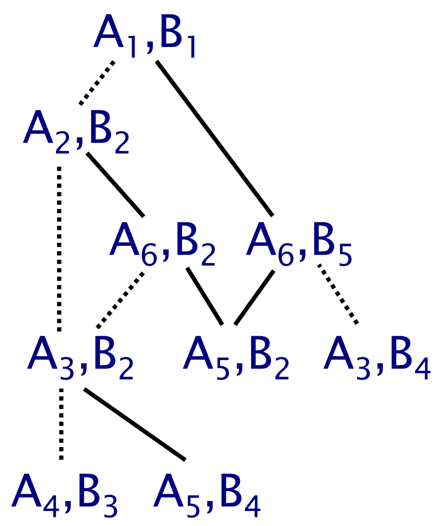
Apply – Example

- Example: Apply(∨, A, B)

Argument BDDs, with node labels:

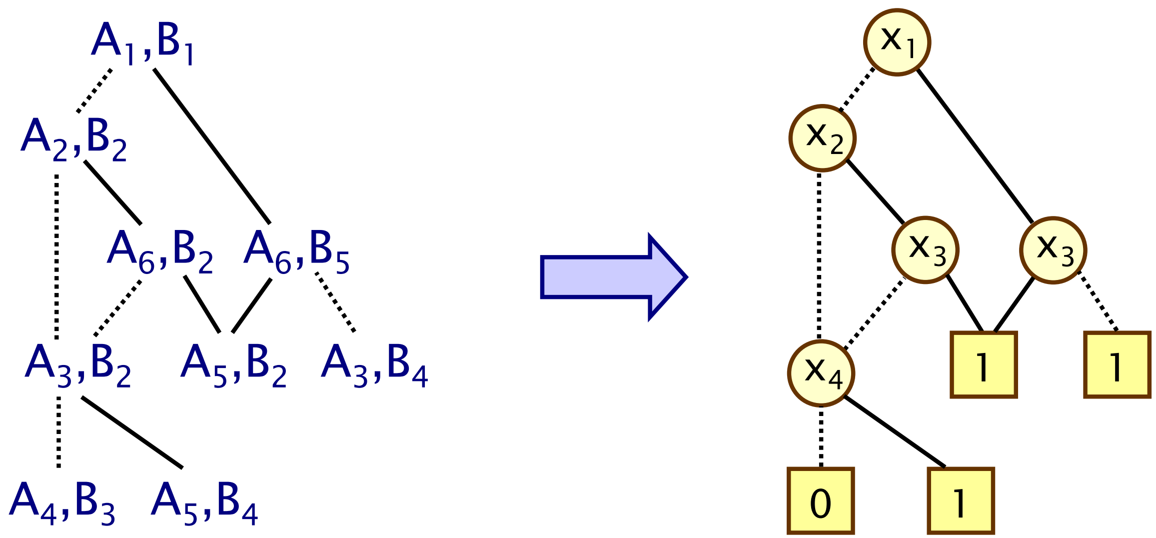


Recursive calls to Apply:



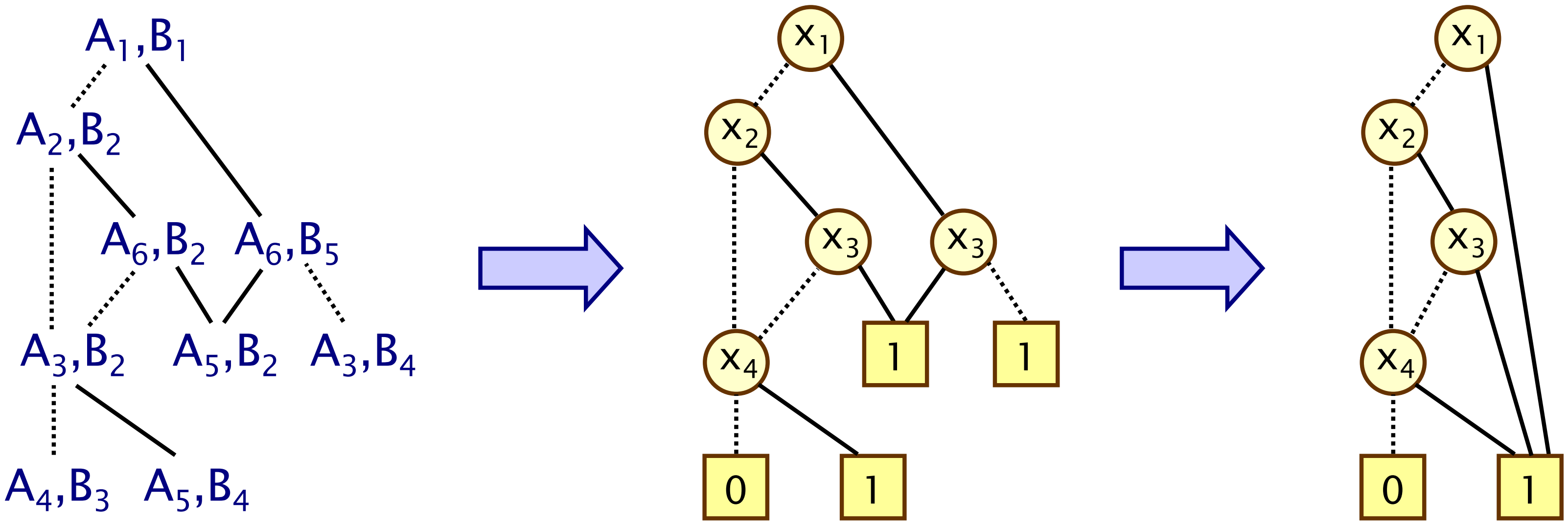
Apply – Example

- Example: Apply(∨, A, B)
  - recursive call structure implicitly defines resulting BDD



Apply – Example

- Example: Apply(∨, A, B)
  - but the resulting BDD needs to be reduced
  - in fact, we can do this as part of the recursive Apply operation, implementing reduction rules bottom-up



Implementation of BDDs

- Store all BDDs currently in use as one multi-rooted BDD
  - no duplicate BDD subtrees, even across multiple BDDs
  - every time a new node is created, check for existence first
  - sometimes called the "unique table"
  - implemented as set of hash tables, one per Boolean variable
  - need: node referencing/dereferencing, garbage collection
- Efficiency implications
  - very significant memory savings
  - trivial checking of BDD equality (pointer comparison)
- Caching of BDD operation results for reuse
  - store result of every BDD operation (memory dependent)
  - applied at every step of recursive BDD operations
  - relies on fast check for BDD equality

Operations with BDDs

- Operations on sets of states easy with BDDs
  - set union: $A \cup B$, in BDDs: $A \vee B$
  - set intersection: $A \cap B$, in BDDs: $A \wedge B$
  - set complement: $S \setminus A$, in BDDs: $\neg A$
- Graph-based algorithms (e.g. reachability)
  - need forwards or backwards image operator
    - i.e. computation of all successors/predecessors of a state
    - again, easy with BDD operations (conjunction, quantification)
  - other ingredients
    - set operations (see above)
    - equality of state sets (fixpoint termination) - equality of BDDs

Summing up…

- Implementation of model checking
  - graph-based algorithms, e.g. reachability, SCC detection
  - manipulation of sets of states, transition relations

- Binary decision diagrams (BDDs)
  - representation for Boolean functions
  - efficient storage/manipulation of sets, transition relations
  - suits symbolic of (e.g.) CTL model checking