UNIVERSITY OF TWENTE.

Formal Methods & Tools.

Scalable Multi-core Model Checking: Technology & Applications of Brute Force
Part III: Symbolic

Jaco van de Pol

30, 31 October 2014

VTSA 2014, Luxembourg

Table of Contents

Sources on Binary Decision Diagrams

Papers/Tutorials (1990's)

- H.R. Andersen, *An Introduction to Binary Decision Diagrams*
- R.E. Bryant, *Symbolic Boolean Manipulation with Ordered Binary-Decision Diagrams*

Tools

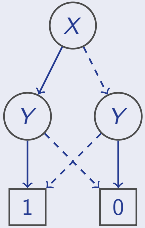
- BDD-packages: BuDDy, CuDD, Java(B)DD, multi-core: Sylvan
- Symbolic model checker: nuSMV `http://nusmv.fbk.eu/`
- LTSmin: `http://fmt.cs.utwente.nl/tools/ltsmin/`

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: $X \Leftrightarrow Y$

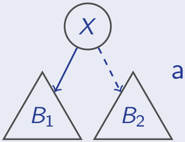
How to interpret a BDD?

Boolean Functions – or sets of Boolean vectors

- 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

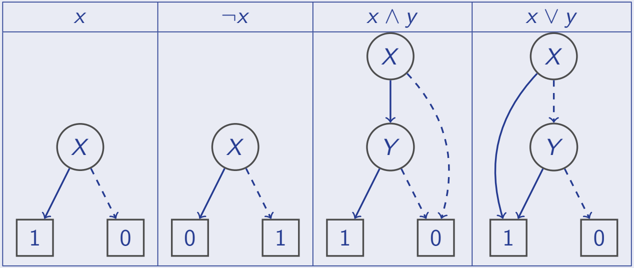


as one of:

- If X then B_1 else B_2. Notation: $X \to B_1, B_2$
- $(X \wedge B_1) \vee (\neg X \wedge B_2)$.

Examples

Basic Boolean Connectives

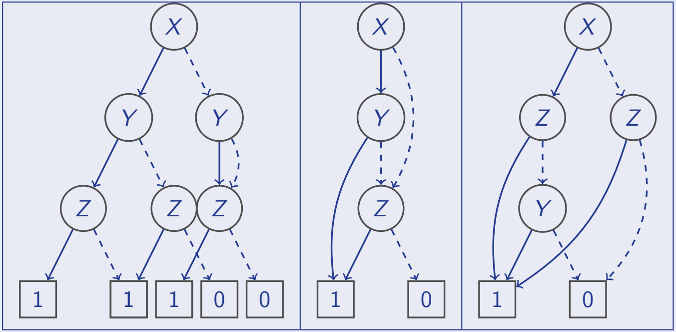


Propositional logic formulas

- 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 one ternary operator: $x \rightarrow s, t$; . . . sufficient basis!

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, $low(v) \neq high(v)$.

Ordered BDDs

A BDD is called ordered iff

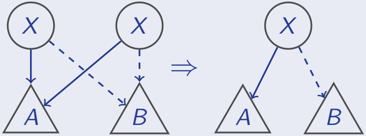
- there exists an ordering $x_1 < x_2 < \cdots < x_n$, such that
- for all nodes v in the BDD,
  $var(v) < var(low(v))$ and $var(v) < var(high(v))$

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

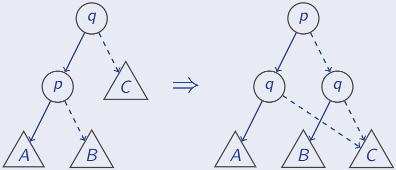
- Eliminate duplicate nodes:



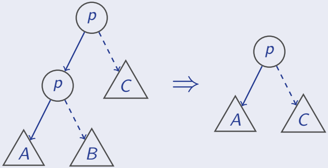
- Eliminate redundant tests:

Stepwise ordering

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



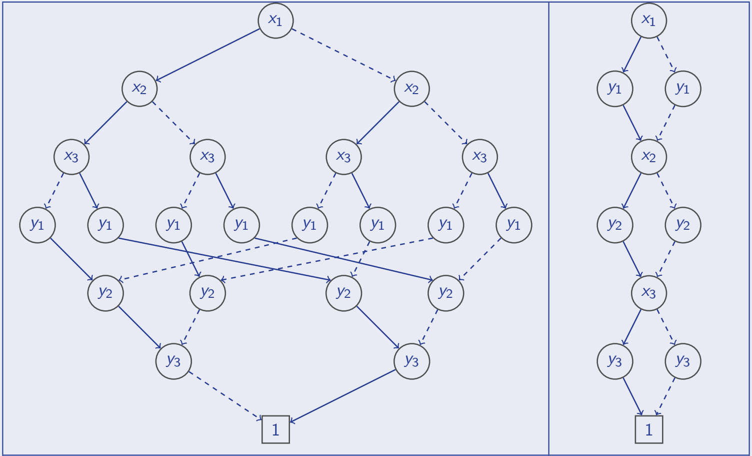
- Eliminate double tests



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)

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



$3.2^n - 2$ 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

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

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:

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

- 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.

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 $\langle x, N_L, N_H \rangle$ 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 $N_1 \wedge N_2$

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**

        $N := \text{MakeNode}(x_1, \text{ApplyAnd}(l_1, l_2), \text{ApplyAnd}(r_1, r_2))$

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

        $N := \text{MakeNode}(x_1, \text{ApplyAnd}(l_1, N_2), \text{ApplyAnd}(r_1, N_2))$

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

        $N := \text{MakeNode}(x_2, \text{ApplyAnd}(N_1, l_2), \text{ApplyAnd}(N_1, r_2))$

    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

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, e.g. 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
- So with dynamic programming, APPLY takes $\mathcal{O}(m \cdot n)$ time

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

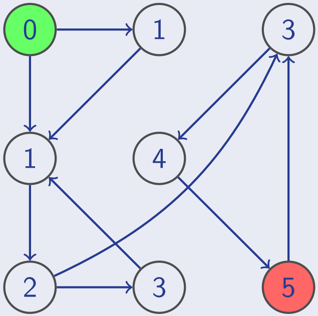
And what about the user?

- 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

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 $\notin$ Error
Queue := [init]
Visited := {init}
while Queue $\neq$ [] **do**
    pick s from front of Queue
    for all t with $s \rightarrow t$ **do**
        if $t \notin$ Visited **then**
            check that $t \notin$ Error
            put t to the end of Queue
            add t to Visited
        end if
    end for
end while

Set-based BFS (variant 1)

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

Set-based BFS (variant 2)

$V_{\text{old}} := \emptyset$
$V_{\text{new}} := \{\text{init}\}$
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

Table of Contents

Kripke structures

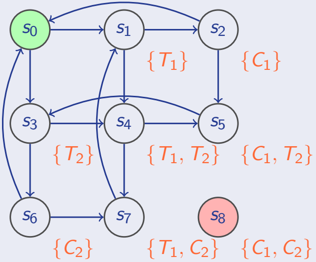
Definition

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

Example of Kripke Structure

Mutual Exclusion / Critical Section



Parts of the Kripke Structure tuple:

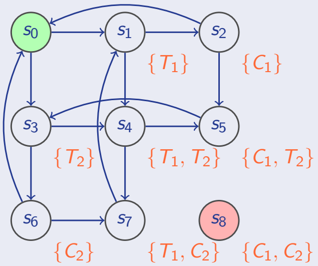
- States $S = \{s_0, \ldots, s_8\}$; Initial states $S_0 = \{s_0\}$.
- $R = \{(s_0, s_1), (s_1, s_2), \ldots\}$
- $AP = \{T_1, C_1, T_2, C_2\}$ (trying, critical)
- $L(s_0) = \varnothing$; $L(s_4) = \{T_1, T_2\}$; $L(s_7) = \{T_1, C_2\}, \ldots$

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

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



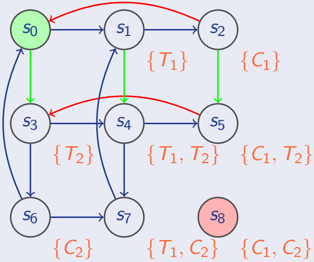
- States correspond to formulas:
  $s_0 = \neg T_1 \wedge \neg T_2 \wedge \neg C_1 \wedge \neg C_2$
  $s_4 = T_1 \wedge T_2 \wedge \neg C_1 \wedge \neg C_2$
- Set of states are also formulas:
  $\{s_1, s_4, s_7\} = T_1$
  $\{s_6, s_7, s_8\} = C_2$
- Set of reachable states?
  $\neg(C_1 \wedge C_2) \wedge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

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

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: $\big(\exists \vec{x}.\, P(\vec{x}) \wedge Q(\vec{x}, \vec{x'})\big)[\vec{x}/\vec{x'}]$

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)

$Vis := Cur := \{\text{init}\}$
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

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 $R = R_1 \vee \cdots \vee R_n$

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'
- Or: dynamically reorder variables during computation

Symbolic Reachability algorithm, revisited

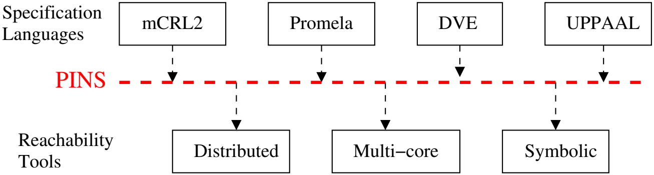
Reachable(I, N, R_i) $= V_{\text{new}}$

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**
  $V_{\text{old}} := V_{\text{new}}$
  for $i = 1$ to N **do**
    $V_{\text{new}} := BDDapply(\vee, V_{\text{new}}, BDDrelprod(V_{\text{new}}, R_i))$
  end for
end while

Table of Contents

Recall LTSmin architecture and PINS interface



	x	y	z
t_1	r	w	–
t_2	–	r	w
t_3	w	–	rw

Advantages of tool and interface (LTSmin / PINS)

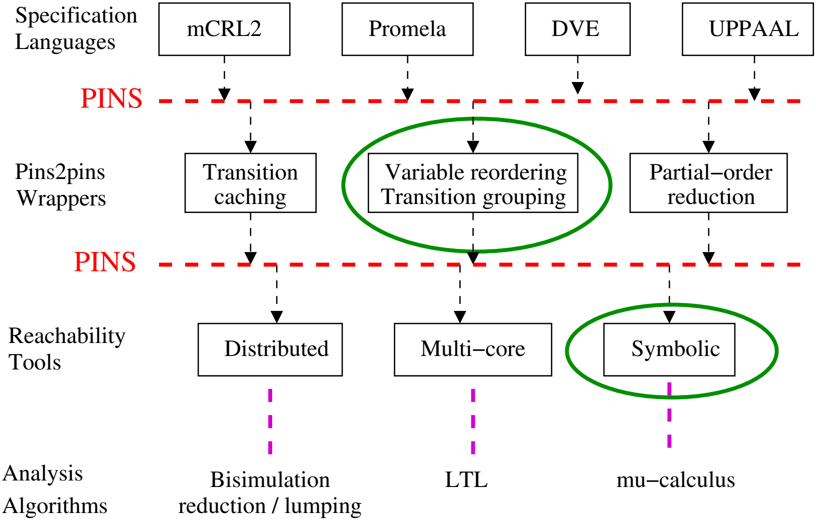
- General and flexible: support for arbitrary state/edge labels
  - Also: LLVM, parity games, Markov Automata, C-code, B||CSP
  - Indirectly: GSPN, xUML, Signalling Networks in Biology
- On-the-fly API: next-state function to pull the implicit graph
- Efficiency: models expose locality in a dependency matrix

- How to do symbolic model checking on-the-fly?

LTSmin architecture and PINS interface

Blom, van de Pol, Weber [CAV'10], Laarman, van de Pol, Weber [NFM'11]

http://fmt.cs.utwente.nl/tools/ltsmin/



Example Dependency Matrix

```
int x=7;
process p1() {
do
::{x>0 => x--;y++}
::{x>0 => x--;z++}
od }
```

```
int y=3;
process p2() {
do
::{y>0 => y--;x++}
::{y>0 => y--;z++}
od }
```

```
int z=9;
process p3() {
do
::{z>0 => z--;x++}
::{z>0 => z--;y++}
od }
```

Default Matrix

$$\begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} p1 \\ p2 \\ p3 \end{array} & \begin{bmatrix} + & + & + \\ + & + & + \\ + & + & + \end{bmatrix} \end{array}$$

Better Matrix

$$\begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} p1.1 \\ p1.2 \\ p2.1 \\ p2.2 \\ p3.1 \\ p3.2 \end{array} & \begin{bmatrix} + & + & - \\ + & - & + \\ + & + & - \\ - & + & + \\ + & - & + \\ - & + & + \end{bmatrix} \end{array}$$

init state $= \langle 7, 3, 9 \rangle$

$\langle 7,3,9 \rangle \xrightarrow{p1.1} \langle 6,4,9 \rangle$

$\langle 7,3,* \rangle \xrightarrow{p1.1} \langle 6,4,* \rangle$

$\langle 7,3,9 \rangle \xrightarrow{p3.2} \langle 7,4,8 \rangle$

$\langle *,3,9 \rangle \xrightarrow{p3.2} \langle *,4,8 \rangle$

Static Regrouping

Caching Local Transitions

- Consider local transition in specification:

```
p3.2: atomic { z>0 -> z--; y++ }
```

- Dependency matrix row: $\begin{array}{cc} & \texttt{x} \quad \texttt{y} \quad \texttt{z} \\ p3.2 & \begin{bmatrix} 0 & 1 & 1 \end{bmatrix} \end{array}$
- Define projection: $\pi_{p3.2}\langle x, y, z\rangle = \langle y, z\rangle$

Next, consider two consecutive calls to p3.2

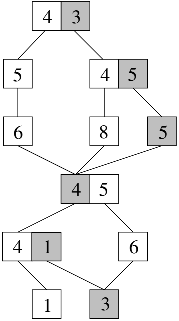
first call:	$\langle x, y, z\rangle$	second call:	$\langle x'', y, z\rangle$
successor:	$\langle x, y', z'\rangle$	project:	$\langle y, z\rangle$
project and	$\langle y, z\rangle \rightarrow$	cache lookup:	$\rightarrow \langle y', z'\rangle$
store in cache:	$\langle y', z'\rangle$	expand:	$\langle x'', y', z'\rangle$

- Transition caching saves calls to the language module
- Memoization table $cache[i]$ for each transition group i

Table of Contents

Multi-valued Decision Diagrams

4	5	6	4	4	1
3	4	8	4	4	1
3	5	5	4	4	1
4	5	6	4	1	3
3	4	8	4	1	3
3	5	5	4	1	3
4	5	6	5	6	3
3	4	8	5	6	3
3	5	5	5	6	3



- Every path in the MDD represents a concrete state vector
- Potential gain in memory saving: exponential (here: 54 $\rightarrow$ 15)
- Symbolic Reachability: explore sets of states stored as MDDs

On-the-fly Symbolic Reachability

- L, V: MDDs for sets of long state vectors (level, visited)
- R_i: MDDs to store transition relation i on short vectors
- L_i, V_i: MDDs for sets of short state vectors (level,visited for i)

symbolic-reachability: learning transition relation

(1) $L := \{\textsc{InitState}()\}$; $V := L$; all $R_i := \emptyset$; all $V_i := \emptyset$
(2) while $L \neq \emptyset$ do
(3) for $i \in groups$ do
(4) $L_i := \pi_i([D]_{N\times K}, L) \setminus V_i$; $V_i := V_i \cup L_i$
(5) $R_i := R_i \cup \{(s, s') \mid s \in L_i \wedge s' \in \textsc{NextState}(i, s)\}$
(6) $L := \bigcup_i(\textsc{RelProd}(R_i, L) \setminus V)$; $V := V \cup L$
(7) return V

- InitState and NextState come from PINS
- RelProd, Or and Diff are MDD operations

Symbolic Reachability

Symbolic Reachability with PINS

- Global set of reachable states is computed as fix point
- Stored as a multi-valued decision diagram (MDD)
- Learn symbolic sub-groups R_i on-the-fly (via NEXTSTATE)

Extensions

- Multiple exploration strategies:
  - Breadth-first: $(T_1 + T_2 + \cdots + T_n)^*$
  - Chaining: $(T_1 \circ T_2 \circ \cdots \circ T_n)^*$
  - Saturation: $((((T_1^* \circ T_2)^* \circ T_3)^* \cdots)^* \circ T_n)^*$
- Static variable reordering boosts performance
- Multiple BDD packages: BuDDy, ListDD, Sylvan (later)

Enjoy playing

- Now $> 2^{50}$ states in seconds, also for PROMELA and mCRL2
- Can connect to any other explicit-state language (fix state vector, split next-state function, dependency matrix)
- mCRL2 example: `xUML Interlocking model`
  - `level 16 has 165499132954291200000 states (7286 nodes)` (so we switched to `gmp-bignum`: $1.6 \cdot 10^{20}$ states)
- Promela example: `GARP` protocol [Igor Konnov, Vienna]:
  - LTSmin explored $3 \cdot 10^{11}$ states in < 3 min using < 300MB.
- Indication for success: sparse dependency matrix

Table of Contents

Earlier work: disappointing speedups

Earlier work in parallel BDDs

- early '90s: vector machines, massive SIMD (not unlike GPU)
- late '90s: virtual SMP, distributed BDDs (BDDnow)

Earlier work in parallel symbolic model checking

- '00vv: Grumberg et al: vertical splitting of BDDs (distributed)
- '00vv: Ciardo et al: horizontal splitting of BDDs (distributed)
- CAV'07: Lüttgen et al: - parallelisation of saturation with Cilk
- PDMC'09: Ciardo - Difficult, but what is the alternative?

Recent developments

- 2010: J. Ossowski - Jinc: Multi-threaded decision diagrams
- 2012: Tom van Dijk, Alfons Laarman, JvdP: Sylvan, a library for multi-core BDD operations (PDMC'12)

Symbolic Model Checking based on BDDs

Bryant [IEEE Trans. Comp.'86], Burch, Clarke, McMillan [LICS'90]

BDD data structures

- Unique Table (to store BDD nodes)
- Computed Cache (apply operations)

Symbolic Reachability (chaining strategy)

Require: I: initial state, $R_1, \ldots, R_N$: subtransitions

Ensure: V_{new}: set of reachable states from I by $\bigcup R_i$

$V_{\text{old}} := \boxed{0}$; $V_{\text{new}} := I$

while $V_{\text{old}} \neq V_{\text{new}}$ **do**

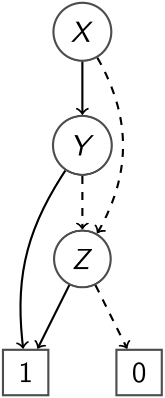
  $V_{\text{old}} := V_{\text{new}}$

  for $i = 1$ to N **do**

    $V_{\text{new}} := V_{\text{new}}$ Or RelProd(V_{new}, R_i)

  end for

end while



Multi-core Binary Decision Diagrams

TOM VAN DIJK, ALFONS LAARMAN, VAN DE POL [PDMC'12]

Multi-core BDDs

- Use shared hashtable for Unique Table, Operations Cache
- Parallelize computation tree of recursive operations (APPLY)

Complications for Parallelism

- BDD nodes can be removed
- Irregular task graph
- Fine-grained parallelism

Solutions

- Tombstones, garbage collect
- Work-stealing
- Use split deque

Table of Contents

Reuse the Lockless Concurrent Hash Table?

Data structures for Binary Decision Diagrams

- Unique Table: ensure that BDD nodes exist at most once
  - implements the maximal sharing requirement
  - necessary for constant equality check by pointer comparison
- Computed Table: dynamic programming
  - Used to store intermediate results to avoid recomputations
  - Needed to manipulate BDDs in polynomial time

- Both data structures are usually implemented as hash tables
- But: Unique Table requires garbage collection in practice

Algorithm to put an entry in Computed Table

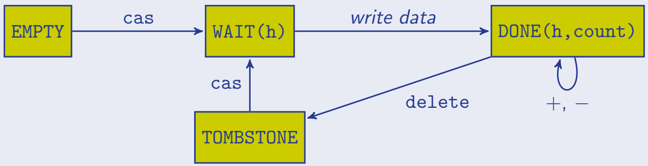
- Local bit-lock in hash array controls access to data array
- Every cache-hit is nice, but don't lose time
  - No waiting for locks: just give up
  - No hash collision resolution: just overwrite or give up

Input: key, data

```
hash ← calculate_hash(key)
index ← hash % tablesize
⟨curhash, curlock⟩ ← hasharray[index]
if curlock = 1 then return NOTADDED
if curhash = hash then
    if data matches data in data array then return NOTADDED
end if
if not compare_and_swap(hasharray[index], ⟨curhash, 0⟩, ⟨hash, 1⟩) then
    return NOTADDED
end if
write data to data array
hasharray[index] ← ⟨hash, 0⟩ {release lock}
return ADDED
```

Unique table = lockless hashtable + garbage collection

- Use one bit of hash as lock, and two bytes as reference count
- Buckets in the hash table are in one of these states:
  1. `EMPTY`: Unused bucket, also end-of-search
  2. `TOMBSTONE`: Unused bucket, but keep searching
  3. `WAIT`(hash): Bucket being written
  4. `DONE`(hash, count): Filled bucket



- Rules to enforce correctness:
  1. Inserting and deleting mutually exclusive (separate gc mode)
  2. Transitions to `WAIT` use `compare_and_swap`

lookup_or_insert algorithm

First part of the algorithm: loop over the *probe sequence* to find existing data.

Input: data

```
hash ← calculate_hash(data)
for i ∈ probe_sequence(data) do
    if bucket[i] = EMPTY then break
    if bucket[i] = WAIT(hash) or bucket[i] = DONE(hash, ∗) then
        while bucket[i] = WAIT(hash) do nothing
        if data matches data in data array then
            increase(bucket[i])
            return i
        end if
    end if
end for
```

lookup_or_insert algorithm

Second part of the algorithm: insert the data!

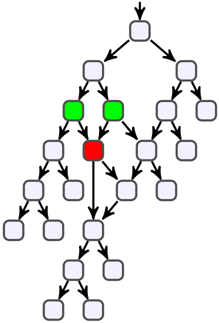
```
for i ∈ probe_sequence(data) do
    value ← bucket[i]
    if value = EMPTY or value = TOMBSTONE then
        if compare_and_swap(bucket[i], value, WAIT(hash)) then
            write data to data array at i
            bucket[i] ← DONE(hash, 1) {release lock and set count to 1}
            return i
        end if
    end if
    if bucket[i] = WAIT(hash) or bucket[i] = DONE(hash, ∗) then
        while bucket[i] = WAIT(hash) do nothing
        if data matches data in data array then
            increase(bucket[i])
            return i
        end if
    end if
end for
return FULL
```

Table of Contents

Parallelizing the BDD Operations

Parallel BDD operations

- Organize recursive calls to RelProd and Or in a *task dependency graph*
- Same task might be created several times: store result in the shared Computed Table
- Fine-grained task-parallelism:
  - Parent spawns children for subtasks, and waits upon their completion
  - Load balancing by work-stealing; use e.g. Cilk [Blumofe '95] or Wool [Faxén '08]



Split double-ended queue in public and private part

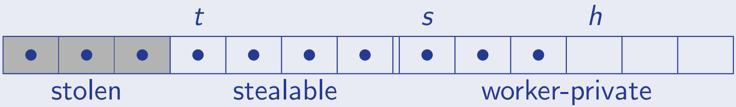


Table of Contents

Recent extensions

- Parallelize Symbolic Reachability itself: more parallel work
  - Parallel learning for all transitions
  - Parallel enumeration of states
  - Parallel RelProd for all transitions
  - Map-reduce type of union of results
- Redesign of Unique Table, mark-and-sweep garbage collection
- Implement parallel MDDs by List Decision Diagrams

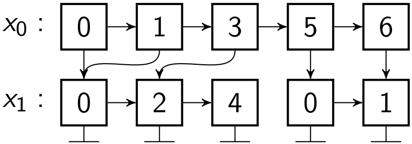
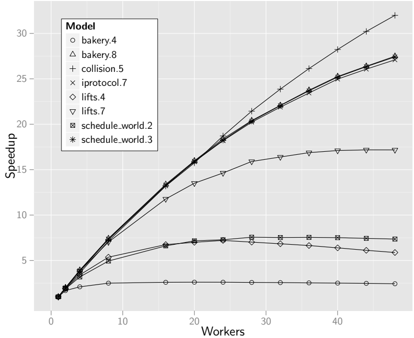


Table of Contents

Results: speedup of BDD operations for model checking



Experiments

- BEEM benchmarks, again
- On $4 \times 12 = 48$ core NUMA
- Speedup up to 32 (=66.7%)
- Small models don't scale (time spent in work stealing)

Conclusion

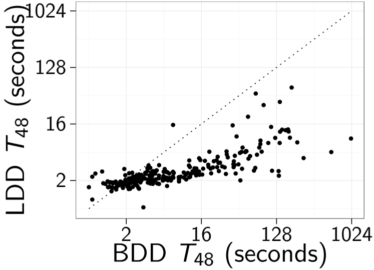
- So far only speed up for the BDD-operations
- Even for large models, many small BDDs are involved

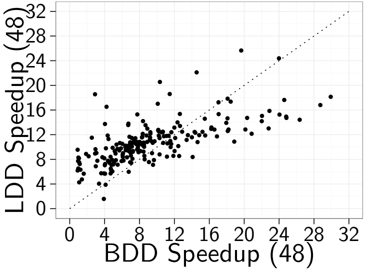
Parallel learning/Reachability vs Parallel BDDs only

High-level parallelism

Parallel BDDs only

Comparing multicore BDDs with multicore MDDs





Literature on LTSmin (symbolic reachability)

LTSmin toolset: symbolic reachability

- http://fmt.cs.utwente.nl/tools/ltsmin/
- Stefan Blom, Jaco van de Pol, ... (ICTAC 2008)
  Symbolic reachability for process algebras with recursive data types
- Stefan Blom, Jaco van de Pol, Michael Weber,
  LTSmin: Distributed and Symbolic Reachability .... (CAV 2010)
- Jeroen Meijer, Gijs Kant, Stefan Blom, Jaco van de Pol .... (HVC 2014)
  Read, Write and Copy Dependencies for Symbolic Model Checking

Multi-core BDDs

- Tom van Dijk, Alfons Laarman and Jaco van de Pol,
  Multi-core BDD Operations for Symbolic Reachability .... (PDMC 2012)
- Tom van Dijk, Jaco van de Pol, ... (MuCoCoS 2014)
  Lace: non-blocking split deque for work-stealing