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

Jaco van de Pol
30, 31 October 2014
Table of Contents

1 Binary Decision Diagrams - storing vectors of sets concisely
   - Definition
   - Implementation

2 Symbolic Reachability
   - Next-state by Relational Product
   - Partitioning of next-state

3 Symbolic Model Checking in LTSmin
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4 Sylvan: Multi-core BDDs
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
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
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:

- 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: $X \iff Y$
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:** $X \to B_1, B_2$
- $(X \land B_1) \lor (\neg X \land B_2)$. 

---

[UNIVERSITY OF TWENTE. Multi-core Model Checking 30, 31 October 2014]
Examples

Basic Boolean Connectives

<table>
<thead>
<tr>
<th>$x$</th>
<th>$\neg x$</th>
<th>$x \land y$</th>
<th>$x \lor y$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Propositional logic formulas

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

Three times: \((x \land y) \lor z\)

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

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 $\text{var}(v) = \text{var}(w)$, $\text{low}(v) = \text{low}(w)$ and $\text{high}(v) = \text{high}(w)$, then $v = w$.
- No redundant tests: For all nodes $v$, $\text{low}(v) \neq \text{high}(v)$.

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, \[ \text{var}(v) < \text{var}(\text{low}(v)) \text{ and } \text{var}(v) < \text{var}(\text{high}(v)) \]
A BDD can (in principle) be transformed to an OBDD by repeated application of the following transformation rules:

### Stepwise reduction

- **Eliminate duplicate nodes:**
  
  \[
  \begin{align*}
  X & \quad X \\
  A & \quad B \\
  \Rightarrow & \\
  X & \\
  A & \quad B
  \end{align*}
  \]

- **Eliminate redundant tests:**
  
  \[
  \begin{align*}
  X & \\
  A & \\
  \Rightarrow & \\
  A
  \end{align*}
  \]

### Stepwise ordering

- **Re-order nodes \((p < q)\):**
  
  \[
  \begin{align*}
  q & \quad p & \quad C \\
  A & \quad B & \quad C \\
  \Rightarrow & \\
  q & \quad p & \quad C \\
  A & \quad B & \quad C
  \end{align*}
  \]

- **Eliminate double tests:**
  
  \[
  \begin{align*}
  p & \quad C \\
  A & \quad B \\
  \Rightarrow & \\
  p & \quad C \\
  A & \quad C
  \end{align*}
  \]
Variable ordering can make an exponential difference

\[(x_1 \Leftrightarrow y_1) \land (x_2 \Leftrightarrow y_2) \land (x_3 \Leftrightarrow y_3)\]  
(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

\[
(x_1 \ldots x_n) \ast (y_1 \ldots y_n) = (z_1 \ldots z_{2n})
\]

needs $O(2^n)$ OBDD nodes, whatever ordering is chosen
- In practice, many functions have small OBDD representations
Table of Contents

1. Binary Decision Diagrams - storing vectors of sets concisely
   - Definition
   - Implementation

2. Symbolic Reachability
   - Next-state by Relational Product
   - Partitioning of next-state

3. Symbolic Model Checking in LTSmin
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4. Sylvan: Multi-core BDDs
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
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:

<table>
<thead>
<tr>
<th>Illustration</th>
<th>(5 &lt; 100 functions in C-interface of BuDDy)</th>
</tr>
</thead>
<tbody>
<tr>
<td>BDD</td>
<td>bdd_high (BDD r)</td>
</tr>
<tr>
<td>BDD</td>
<td>bdd_not (BDD r)</td>
</tr>
<tr>
<td>BDD</td>
<td>bdd_apply (BDD l, BDD r, int op)</td>
</tr>
<tr>
<td>BDD</td>
<td>bdd_exist (BDD r, BDD var)</td>
</tr>
<tr>
<td>BDD</td>
<td>bdd_relprod (BDD l, BDD r, BDD var)</td>
</tr>
</tbody>
</table>

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: \( N \leftrightarrow \langle \text{var}, N_L, N_H \rangle \)
- One can lookup \( \text{var}(N), \text{low}(N), \text{high}(N) \) in \( O(1) \) 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 \((\neg x \land N_L) \lor (x \land N_H)\)

1: if \( N_L = N_H \) then
2: \( N := N_L \)
3: else if \( \langle x, N_L, N_H \rangle \) is in the unique table then
4: \( N := \text{lookup}(x, N_L, N_H) \)
5: else
6: \( N := \text{insert\_new\_entry}(x, N_L, N_H) \) in the unique table
7: end if
Naive function for conjunction: $\wedge$

ApplyAnd($N_1, N_2$) = $N$

**Require:** BDD nodes $N_1, N_2$

**Ensure:** BDD node $N$ representing $N_1 \land N_2$

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
4. $x_1, l_1, r_1 := \text{var}(N_1), \text{low}(N_1), \text{high}(N_1)$
5. $x_2, l_2, r_2 := \text{var}(N_2), \text{low}(N_2), \text{high}(N_2)$
6. if $x_1 = x_2$ then
7. $N := \text{MakeNode}(x_1, \text{ApplyAnd}(l_1, l_2), \text{ApplyAnd}(r_1, r_2))$
8. else if $x_1 < x_2$ then
9. $N := \text{MakeNode}(x_1, \text{ApplyAnd}(l_1, N_2), \text{ApplyAnd}(r_1, N_2))$
10. else if $x_1 > x_2$ then
11. $N := \text{MakeNode}(x_2, \text{ApplyAnd}(N_1, l_2), \text{ApplyAnd}(N_1, r_2))$
12. end if
13. 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 $O(m \cdot n)$ time
Data structure: Operation Cache

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

1: if \((op, N_1, N_2)\) is in the operation cache then
2: \(N := \text{lookup}(op, N_1, N_2)\)
3: else
4: \(x_1, l_1, r_1 := \text{var}(N_1), \text{low}(N_1), \text{high}(N_1)\)
5: \(x_2, l_2, r_2 := \text{var}(N_2), \text{low}(N_2), \text{high}(N_2)\)
6: if \(x_1 = x_2\) then
7: \(N := \text{MakeNode}(x_1, \text{Apply}(op, l_1, l_2), \text{Apply}(op, r_1, r_2))\)
8: else if \(x_1 < x_2\) then
9: \(N := \text{MakeNode}(x_1, \text{Apply}(op, l_1, N_2), \text{Apply}(op, r_1, N_2))\)
10: else if \(x_1 > x_2\) then
11: \(N := \text{MakeNode}(x_2, \text{Apply}(op, N_1, l_2), \text{Apply}(op, N_1, r_2))\)
12: end if
13: add \((op, N_1, N_2) \mapsto N\) to the operation cache
14: 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

1. Binary Decision Diagrams - storing vectors of sets concisely
   - Definition
   - Implementation

2. Symbolic Reachability
   - Next-state by Relational Product
   - Partitioning of next-state

3. Symbolic Model Checking in LTSmin
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4. Sylvan: Multi-core BDDs
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
Breadth First Search, example

### BFS traversal

- Eventually all states will be visited
- Shortest path will be detected
### Explicit-state BFS

1. check that init \( \notin \text{Error} \)
2. Queue := [init]
3. Visited := {init}
4. while Queue \( \neq \) [] do
5. pick s from front of Queue
6. for all t with s \( \rightarrow \) t do
7. if t \( \notin \) Visited then
8. check that t \( \notin \) Error
9. put t to the end of Queue
10. add t to Visited
11. end if
12. end for
13. end while

### Set-based BFS (variant 1)

1. Vis := Cur := {init}
2. while Cur \( \neq \) \( \emptyset \) do
3. check Cur \( \cap \) Error = \( \emptyset \)
4. Cur := Next(Cur, \( \rightarrow \)) \( \setminus \) Vis
5. Vis := Vis \( \cup \) Cur
6. end while

### Set-based BFS (variant 2)

1. \( V_{\text{old}} := \emptyset \)
2. \( V_{\text{new}} := \{\text{init}\} \)
3. while \( V_{\text{old}} \neq V_{\text{new}} \) do
4. \( V_{\text{old}} := V_{\text{new}} \)
5. \( V_{\text{new}} := V_{\text{old}} \cup \text{Next}(V_{\text{old}}, \rightarrow) \)
6. end while
Table of Contents

1 Binary Decision Diagrams - storing vectors of sets concisely
   - Definition
   - Implementation

2 Symbolic Reachability
   - Next-state by Relational Product
   - Partitioning of next-state

3 Symbolic Model Checking in LTSmin
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4 Sylvan: Multi-core BDDs
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
# 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
- **\(L : S \rightarrow \mathcal{P}(AP)\)** 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)
- \( L(s_0) = \emptyset; L(s_4) = \{T_1, T_2\}; L(s_7) = \{T_1, C_2\}, \ldots \)
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:
  \( s_0 = \neg T_1 \land \neg T_2 \land \neg C_1 \land \neg C_2 \)
  \( s_4 = T_1 \land T_2 \land \neg C_1 \land \neg C_2 \)

- 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?
  \( \neg (C_1 \land C_2) \land \neg (T_1 \land C_1 \lor T_2 \land C_2) \)
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: $\neg T_2 \land \neg C_2 \land T_2' \land \neg C_2'$
  - Red: $C_1 \land \neg C_2 \land \neg C_1' \land \neg T_1'$
- This is not quite true: also ensure other variables don’t change:
  - Green: $\ldots \land T_1 = T_1' \land C_1 = C_1'$
  - Red: $\ldots \land T_2 = T_2' \land C_2 = C_2'$. 
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: $P(\vec{x}) \land Q(\vec{x}, \vec{x}')$
- Step 2: Abstract previous states: $\exists \vec{x}. P(\vec{x}) \land Q(\vec{x}, \vec{x}')$
- Step 3: Rename back to original state variable names: $(\exists \vec{x}. P(\vec{x}) \land Q(\vec{x}, \vec{x}'))[\vec{x}/\vec{x}']$

What about existential quantification?

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

#### Recall Set-BFS (variant 1)

1. \( \text{Vis} := \text{Cur} := \{ \text{init} \} \)
2. while \( \text{Cur} \neq \emptyset \) do  
3. \( \text{check } \text{Cur} \cap \text{Error} = \emptyset \)  
4. \( \text{Cur} := \text{Next}(\text{Cur}, \rightarrow) \setminus \text{Vis} \)  
5. \( \text{Vis} := \text{Vis} \cup \text{Cur} \)  
6. end while

#### Recall Set-BFS (variant 2)

1. \( \text{V}_{\text{old}} := \emptyset \)  
2. \( \text{V}_{\text{new}} := \{ \text{init} \} \)  
3. while \( \text{V}_{\text{old}} \neq \text{V}_{\text{new}} \) do  
4. \( \text{V}_{\text{old}} := \text{V}_{\text{new}} \)  
5. \( \text{V}_{\text{new}} := \text{V}_{\text{old}} \cup \text{Next}(\text{V}_{\text{old}}, \rightarrow) \)  
6. end while

#### Implementation with BDDs

- **Vis, Cur**: Binary Decision Diagrams (BDDs)  
- **∩**: BDDapplyAnd  
- **∪**: BDDapplyOr  
- **Next**: BDDRelProd  
- **≠**: pointer comparison (unique representation!)
Table of Contents

1 Binary Decision Diagrams - storing vectors of sets concisely
   - Definition
   - Implementation

2 Symbolic Reachability
   - Next-state by Relational Product
   - Partitioning of next-state

3 Symbolic Model Checking in LTSmin
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4 Sylvan: Multi-core BDDs
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
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: $R_1 \land R_2 \land \cdots \land R_n$.

A-synchronous systems: disjunctive partitioning

- In parallel software: transitions are interleaved, non-determinism
- So the relation of the system is: $R_1 \lor R_2 \lor \cdots \lor R_n$.
- (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 $R = R_1 \lor \cdots \lor R_n$

**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:** \(V_{\text{new}}\) BDD representing the reachable states from \(I\) by \(R_i\)

\[
\begin{align*}
V_{\text{old}} & := \text{BDDempty} \\
V_{\text{new}} & := I \\
\text{while} & \text{ not } \text{BDDequal}(V_{\text{old}}, V_{\text{new}}) \text{ do} \\
& \quad V_{\text{old}} := V_{\text{new}} \\
& \quad \text{for } i = 1 \text{ to } N \text{ do} \\
& \quad \quad V_{\text{new}} := \text{BDDapply}(\lor, V_{\text{new}}, \text{BDDrelprod}(V_{\text{new}}, R_i)) \\
& \quad \text{end for} \\
& \text{end while}
\end{align*}
\]
Table of Contents

1. Binary Decision Diagrams - storing vectors of sets concisely
   - Definition
   - Implementation

2. Symbolic Reachability
   - Next-state by Relational Product
   - Partitioning of next-state

3. Symbolic Model Checking in LTSmin
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4. Sylvan: Multi-core BDDs
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
Recall LTSmin architecture and PINS interface

Specification Languages
- mCRL2
- Promela
- DVE
- UPPAAL

PINS
- Distributed
- Multi-core
- Symbolic

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>t₁</td>
<td>r</td>
<td>w</td>
<td>–</td>
</tr>
<tr>
<td>t₂</td>
<td>–</td>
<td>r</td>
<td>w</td>
</tr>
<tr>
<td>t₃</td>
<td>w</td>
<td>–</td>
<td>rw</td>
</tr>
</tbody>
</table>

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/

Specification Languages
- mCRL2
- Promela
- DVE
- UPPAAL

PINS

Pins2pins Wrappers
- Transition caching
- Variable reordering
- Transition grouping
- Partial-order reduction

Reachability Tools
- Distributed
- Multi-core
- Symbolic

Analysis Algorithms
- Bisimulation reduction / lumping
- LTL
- mu-calculus

UNIVERSITY OF TWENTE.
### Example Dependency Matrix

```plaintext
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 }
```

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>p1</strong></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>p2</strong></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>p3</strong></td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

**Default Matrix**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>p1.1</strong></td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>p1.2</strong></td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td><strong>p2.1</strong></td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>p2.2</strong></td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>p3.1</strong></td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td><strong>p3.2</strong></td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

**Better Matrix**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>p1.1</strong></td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>p1.2</strong></td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td><strong>p2.1</strong></td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td><strong>p2.2</strong></td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td><strong>p3.1</strong></td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td><strong>p3.2</strong></td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

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

\begin{align*}
\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
\end{align*}

**Static Regrouping**
Caching Local Transitions

- Consider local transition in specification:
  \[ p3.2: \text{atomic} \{ \text{z}>0 \rightarrow \text{z--}; \text{y++} \} \]

- Dependency matrix row:
  \[
  \begin{bmatrix}
  x & y & z \\
  p3.2 & 0 & 1 & 1
  \end{bmatrix}
  \]

- Define projection: \( \pi_{p3.2}(x, y, z) = (y, z) \)

Next, consider two consecutive calls to \( p3.2 \)

<table>
<thead>
<tr>
<th>first call: ( (x, y, z) )</th>
<th>second call: ( (x'', y, z) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>successor: ( (x, y', z') )</td>
<td>project: ( (y, z) )</td>
</tr>
<tr>
<td>project and store in cache: ( (y, z) \rightarrow (y', z') )</td>
<td>cache lookup: ( \rightarrow (y', z') )</td>
</tr>
<tr>
<td>expand: ( (x'', y', z') )</td>
<td>expand: ( (x'', y', z') )</td>
</tr>
</tbody>
</table>

- Transition caching saves calls to the language module
- Memoization table \( cache[i] \) for each transition group \( i \)
Table of Contents

1 Binary Decision Diagrams - storing vectors of sets concisely
   - Definition
   - Implementation

2 Symbolic Reachability
   - Next-state by Relational Product
   - Partitioning of next-state

3 Symbolic Model Checking in LTSmin
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4 Sylvan: Multi-core BDDs
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
Multi-valued Decision Diagrams

- Every path in the MDD represents a concrete state vector
- Potential gain in memory saving: exponential (here: 54 → 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

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

- **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)
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

1. Binary Decision Diagrams - storing vectors of sets concisely
   - Definition
   - Implementation

2. Symbolic Reachability
   - Next-state by Relational Product
   - Partitioning of next-state

3. Symbolic Model Checking in LTSmin
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4. Sylvan: Multi-core BDDs
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
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)
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: $V_{\text{new}}$: set of reachable states from $I$ by $\bigcup R_i$

1. $V_{\text{old}} := 0$; $V_{\text{new}} := I$
2. while $V_{\text{old}} \neq V_{\text{new}}$ do
3. $V_{\text{old}} := V_{\text{new}}$
4. for $i = 1$ to $N$ do
5. $V_{\text{new}} := V_{\text{new}} \text{ Or } \text{RelProd}(V_{\text{new}}, R_i)$
6. end for
7. 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

1 Binary Decision Diagrams - storing vectors of sets concisely
   - Definition
   - Implementation

2 Symbolic Reachability
   - Next-state by Relational Product
   - Partitioning of next-state

3 Symbolic Model Checking in LTSmin
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4 Sylvan: Multi-core BDDs
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
The slide titled "Reuse the Lockless Concurrent Hash Table?" contains the following content:

## 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

1: hash ← calculate_hash(key)
2: index ← hash % tablesize
3: \langle curhash, curlock \rangle ← hasharray[index]
4: if curlock = 1 then return NOTADDED
5: if curhash = hash then
6: if data matches data in data array then return NOTADDED
7: end if
8: if not compare_and_swap(hasharray[index], \langle curhash, 0 \rangle, \langle hash, 1 \rangle) then
9: return NOTADDED
10: end if
11: write data to data array
12: hasharray[index] ← \langle hash, 0 \rangle \{release lock\}
13: 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
1: hash ← calculate_hash(data)
2: for i ∈ probe_sequence(data) do
3:     if bucket[i] = EMPTY then break
4:     if bucket[i] = WAIT(hash) or bucket[i] = DONE(hash, *) then
5:         while bucket[i] = WAIT(hash) do nothing
6:     if data matches data in data array then
7:         increase(bucket[i])
8:         return i
9:     end if
10:    end if
11:   end for
```
lookup_or_insert algorithm

Second part of the algorithm: insert the data!

```
9: for i ∈ probe_sequence(data) do
10:   value ← bucket[i]
11:   if value = EMPTY or value = TOMBSTONE then
12:     if compare_and_swap(bucket[i], value, WAIT(hash)) then
13:       write data to data array at i
14:     bucket[i] ← DONE(hash, 1) \{release lock and set count to 1\}
15:     return i
16:   end if
17: end if
18: if bucket[i] = WAIT(hash) or bucket[i] = DONE(hash, *) then
19:   while bucket[i] = WAIT(hash) do nothing
20: if data matches data in data array then
21:   increase(bucket[i])
22:   return i
23: end if
24: end if
25: end for
26: return FULL
```
# Table of Contents

1. **Binary Decision Diagrams - storing vectors of sets concisely**
   - Definition
   - Implementation

2. **Symbolic Reachability**
   - Next-state by Relational Product
   - Partitioning of next-state

3. **Symbolic Model Checking in LTSmin**
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4. **Sylvan: Multi-core BDDs**
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
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>
<thead>
<tr>
<th>t</th>
<th>s</th>
<th>h</th>
</tr>
</thead>
<tbody>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

stolen stealable worker-private
Table of Contents

1. Binary Decision Diagrams - storing vectors of sets concisely
   - Definition
   - Implementation

2. Symbolic Reachability
   - Next-state by Relational Product
   - Partitioning of next-state

3. Symbolic Model Checking in LTSmin
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4. Sylvan: Multi-core BDDs
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
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

\[ \begin{align*}
  x_0 : &\quad 0 \rightarrow 1 \rightarrow 3 \rightarrow 5 \rightarrow 6 \\
  x_1 : &\quad 0 \rightarrow 2 \rightarrow 4 \rightarrow 0 \rightarrow 1
\end{align*} \]
## Table of Contents

1. **Binary Decision Diagrams - storing vectors of sets concisely**
   - Definition
   - Implementation

2. **Symbolic Reachability**
   - Next-state by Relational Product
   - Partitioning of next-state

3. **Symbolic Model Checking in LTSmin**
   - PINS interface and Local Transition Caching
   - Multi-valued Decision Diagrams

4. **Sylvan: Multi-core BDDs**
   - Data Structures
   - Work Stealing
   - Parallelism at a higher level
   - Experiments
Results: speedup of BDD operations for model checking

Experiments

- BEEM benchmarks, again
- On 4 × 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

![Comparison of BDDs and LDDs](chart.png)

- **BDD**
  - Time (seconds)
- **LDD**
  - Time (seconds)
- **BDD Speedup (48)**
  - Time (seconds)
- **LDD Speedup (48)**
  - Time (seconds)
Literature on LTSmin (symbolic reachability)

LTSmin toolset: symbolic reachability

- 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