Data Decision Diagrams, Set Decision Diagrams, And Applications

Yann Thierry-Mieg
23 Oct 2008
Presentation in Nantes
Decision Diagrams for Model-checking:
- Data Decision Diagrams:
  - *A dynamic and flexible DD library*
- Saturation
  - *A more effective fixpoint strategy*
- Hierarchical Set Decision Diagrams
  - *Introduce structured descriptions*
- Instantiable Transition Systems
  - *A framework to exploit SDD*
Data Decision Diagrams
ICATPN'2002

J.M. Couvreur, P.A. Wacrenier
E. Encrenaz, E. Paviot-Adet, D. Poitrenaud
LIP6, LaBRI

Data Decision Diagrams
Saturation
Set Decision Diagrams
Instantiable Transition System
- Decision diagrams: [BCM'92]
  - Initially BDD [Bryant86]
  - Compact Structure to represent sets
  - Unicity table & operation cache
  - Complexity linked to number of nodes
  - Exploits implicit symmetries between elements of the set
  - Intermediate peak size problem
- Very widespread success
  - SMV, Smart, Uppaal, Prism, ...
- Data Decision Diagram [Couvreur+02]
  - integer domain variables, no ordering, variable length paths
  - Set Operations + inductive homomorphisms

\[
\begin{align*}
  a & \rightarrow c & 1 \rightarrow 1 \\
  a & \rightarrow a & 2 \rightarrow c & 1 \rightarrow 1 \\
  a & \rightarrow a & 1 \rightarrow b & 3 \rightarrow 1
\end{align*}
\]
E = \{ \text{ variable } \}, \text{ Dom}(e) = \text{ domain of } e

Inductive definition

- 0, 1, T are DDDs or
- \( d = (e, \alpha) \) with
  - \( e \) variable in \( E \),
  - \( \alpha : \text{Dom}(e) \to \text{DDD} \)
  - \( |\alpha| < \infty \)

is a DDD

Properties

- No variable order
- A variable may appear twice in a decision path
- A domain may be finite or infinite
- A DDD is a finite structure
- Canonical representation: “if \( \alpha = 0 \) then (\( e, \alpha \)) \equiv 0” \( \Rightarrow \) zero suppressed
d is better defined than d’ iff

• d = d’ or
• d = T or
• d = (e,α), d’=(e,α’) with α(x) is better defined than α’ (x) for any x in Dom(e)

“is better defined than”

DDD operators

A mapping f : DDD^n → DDD on DDD is a **DDD operator** if f is compatible with the better defined relation:

∀i : d_i ≤ d_i’ ⇒ f(d_1, …, d_n) ≤ f(d_1’, …, d_n’)
Union and the undefined terminal T

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>T</th>
<th>((e_2, \alpha_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>T</td>
<td>((e_2, \alpha_2))</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>((e_1, \alpha_1))</td>
<td>((e_1, \alpha_1))</td>
<td>T</td>
<td>T</td>
<td>e_1 = e_2 ? ((e_1, \alpha_1 + \alpha_2) : T)</td>
</tr>
</tbody>
</table>

Union "+"

![Diagram showing the union of two trees representing \((e_1, \alpha_1)\) and \((e_2, \alpha_2)\) with a root node labeled '+'.]
### Union and the undefined terminal T

#### Union "+"

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>1</th>
<th>T</th>
<th>(e₂, α₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>T</td>
<td>(e₂, α₂)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>(e₁, α₁)</td>
<td>(e₁, α₁)</td>
<td>T</td>
<td>T</td>
<td>e₁=e₂ ? (e₁, α₁+α₂) : T</td>
</tr>
</tbody>
</table>

The table above represents the union operation for terminals with labels α₁ and α₂. The symbol T represents the undefined terminal.

The diagrams illustrate the union operation for terminals a and b.

The result of the union is shown on the right side of the equation.

The union operation is denoted by the symbol +.
Union and the undefined terminal T

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>T</th>
<th>((e_2, \alpha_2))</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>T</td>
<td>((e_2, \alpha_2))</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>((e_1, \alpha_1))</td>
<td>((e_1, \alpha_1))</td>
<td>T</td>
<td>T</td>
<td>e_1 = e_2 ? ((e_1, \alpha_1 + \alpha_2) : T)</td>
</tr>
</tbody>
</table>

Union "+"

Diagram of the union operation.
Union and the undefined terminal T

Union "+"

<table>
<thead>
<tr>
<th>+</th>
<th>0</th>
<th>1</th>
<th>T</th>
<th>(e₂, α₂)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>T</td>
<td>(e₂, α₂)</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>T</td>
<td>T</td>
</tr>
<tr>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
<td>T</td>
</tr>
</tbody>
</table>

\[
(e₁, α₁) + (e₁, α₁) = \begin{cases} T & \text{if } e₁ = e₂ \Leftrightarrow (e₁, α₁ + α₂) : T\end{cases}
\]

Danger here!
We avoid T by carefully choosing operands of union.
Other Elementary Operators on DDD

- **Intersection "*", Set difference "/"**
  - defined in a usual manner, with added cases to handle T (not presented here)

- **Concatenation "."**:
  - specific operation to extend paths

\[
\begin{array}{ccc}
\text{a} & . & \text{b} \\
1 & 2 & 3 \\
1 & 1 & 3 \\
\end{array}
\quad \quad \quad = \quad \quad \quad
\begin{array}{ccc}
\text{a} & \text{b} & \text{a} \\
1 & 2 & 2 \\
1 & 1 & 1 \\
\end{array}
\]
Homomorphism

Definition

\[ \Phi : \text{DDD} \rightarrow \text{DDD} \]

\[ \forall d_1, d_2 \]

\[ \begin{cases} 
\Phi(0) = 0 \\
\Phi(d_1) + \Phi(d_2) \leq \Phi(d_1 + d_2) \\
d_1 \leq d_2 \Rightarrow \Phi(d_1) \leq \Phi(d_2) 
\end{cases} \]

Examples, hard coded in the library

\[ \text{Id, Id + d, Id} * d, \text{Id} \setminus d, \text{Id}.d, d.Id \]

Proposition

If \( \Phi_1, \Phi_2 \) are homomorphisms then

\[ \Phi_1 + \Phi_2 \text{ and } \Phi_1 \circ \Phi_2 \text{ are homomorphisms} \]
**Proposition**

Let $(\pi_{i,j})$, $(\tau_i)$ be families of homomorphisms. The mappings $(\Phi_i)$ inductively defined by

\[
\Phi_i(d) = \begin{cases} 
0 & \text{if } d = 0 \\
\Phi_i(1) & \text{if } d = 1 \\
\top & \text{if } d = \top \\
\sum_x \Phi_i(e,x)(\alpha(x)) & \text{if } d = (e,\alpha) 
\end{cases}
\]

are homomorphisms where

\[
\Phi_i(e,x) = \sum_j \pi_{i,j} \circ \Phi_j(e,x) + \tau_i
\]

User defined homomorphisms are defined by:

- $\Phi(1)$, a constant DDD that is the evaluation over terminal 1
- $\Phi(e,x)$, for any variable $e$ and value $x$, a homomorphism to apply on the successor node
Example:
Increment the value of variable \( v \)

\[
Inc(v)(e,x) = \begin{cases} 
  v \overset{x+1}{\to} \text{Id} & \text{if } v = e \\
  e \overset{x}{\to} Inc(v) & \text{otherwise}
\end{cases}
\]

\[
Inc(v)(1) = 1
\]
Example: 
**Increment the value of variable v**

\[
\text{Inc}(v)(e, x) = \begin{cases} 
  v \xrightarrow{x+1} \text{Id} & \text{if } v = e \\
  e \xrightarrow{x} \text{Inc}(v) & \text{otherwise}
\end{cases}
\]

\[
\text{Inc}(v)(1) = 1
\]
\[ \text{Inc}(v)(e,x) = \begin{cases} 
    v \xrightarrow{x+1} \text{Id} & \text{if } v=e \\
    e \xrightarrow{x} \text{Inc}(v) & \text{otherwise} 
\end{cases} \]

\[ \text{Inc}(v)(1) = 1 \]
• **Can be defined compositionnally:**
  • **Simple operations**
    • *hard coded basic operators*
      • \( \text{Id, Id} + \text{d, Id} \times \text{d, Id} \setminus \text{d, Id.d, d.Id} \)
    • *simple homomorphisms like Inc()*
  • **Compose complex operations using**
    • ◦ composition &
    • ∪ union +
    • ∩ intersection *
  • **No set difference**
    • *in general does not preserve linearity*
  • **Use transitive closure \( h^* \)**
    • *newly introduced*
Composition Example:
Swap the values of two variables v, w

Swap(v,w)(e,x) = \begin{cases} 
\text{Rename}(v) \circ \text{Down}(w,x) & \text{if } v = e \\
\text{Rename}(w) \circ \text{Down}(v,x) & \text{if } w = e \\
e \xrightarrow{x} \text{Swap}(v,w) & \text{otherwise}
\end{cases}

\text{Swap}(v,w)(1) = 1

\text{Rename}(v)(e,x) = v \xrightarrow{x} \text{Id}
\text{Rename}(v)(1) = T

\text{Up}(v,z)(e,x) = e \xrightarrow{x} v \xrightarrow{z} \text{Id}
\text{Up}(v,z)(1) = T

\text{Down}(v,z)(e,x) = \begin{cases} 
v \xrightarrow{x} v \xrightarrow{z} \text{Id} & \text{if } v = e \\
\text{Up}(e,x) \circ \text{Down}(v,z) & \text{otherwise}
\end{cases}
\text{Down}(v,z)(1) = T
Composition Example: Swap the values of two variables \( v, w \)

\[
\text{Swap}(v, w)(e, x) = \begin{cases} 
\text{Rename}(v) \circ \text{Down}(w, x) & \text{if } v = e \\
\text{Rename}(w) \circ \text{Down}(v, x) & \text{if } w = e \\
\text{Id} & \text{otherwise}
\end{cases}
\]

\[
\text{Swap}(v, w)(1) = 1
\]

\[
\text{Rename}(v)(e, x) = v \xrightarrow{x} \text{Id}
\]

\[
\text{Rename}(v)(1) = T
\]

\[
\text{Down}(v, z)(e, x) = \begin{cases} 
v \xrightarrow{x} v \xrightarrow{z} \text{Id} & \text{if } v = e \\
\text{Id} & \text{otherwise}
\end{cases}
\]

\[
\text{Down}(v, z)(1) = T
\]

\[
\text{Up}(v, z)(e, x) = \begin{cases} 
v \xrightarrow{x} v \xrightarrow{z} \text{Id} & \text{if } v = e \\
\text{Id} & \text{otherwise}
\end{cases}
\]

\[
\text{Up}(v, z)(1) = T
\]
Composition
Swap the values of two variables v, w

Swap(v,w)(e,x) =

\begin{align*}
\text{Rename}(v) \circ \text{Down}(w,x) & \quad \text{if } v = e \\
\text{Rename}(w) \circ \text{Down}(v,x) & \quad \text{if } w = e \\
e \xrightarrow{X} \text{Swap}(v,w) & \quad \text{otherwise}
\end{align*}

Swap(v,w)(1) = 1

Rename(v)(e,x) = v \xrightarrow{X} \text{Id}

Rename(v)(1) = T

Up(v,z)(e,x) = e \xrightarrow{X} v \xrightarrow{Z} \text{Id}

Up(v,z)(1) = T

\begin{align*}
\text{Down}(v,z)(e,x) &=
\begin{cases}
 v \xrightarrow{X} v \xrightarrow{Z} \text{Id} & \quad \text{if } v = e \\
 \text{Up}(e,x) \circ \text{Down}(v,z) & \quad \text{otherwise}
\end{cases}
\end{align*}

Down(v,z)(1) = T
Composition

Swap the values of two variables v, w

\[
\text{Swap}(v,w)(e,x) = \begin{cases} 
\text{Rename}(v) \circ \text{Down}(w,x) & \text{if } v = e \\
\text{Rename}(w) \circ \text{Down}(v,x) & \text{if } w = e \\
e \xrightarrow{X} \text{Swap}(v,w) & \text{otherwise}
\end{cases}
\]

Swap(v,w) (1) = 1

\[
\text{Rename}(v)(e,x) = v \xrightarrow{X} \text{Id}
\]

Rename(v)(1) = T

\[
\text{Up}(v,z)(e,x) = e \xrightarrow{X} v \xrightarrow{Z} \text{Id}
\]

Up(v,z)(1) = T

\[
\text{Down}(v,z)(e,x) = \begin{cases} 
v \xrightarrow{X} v \xrightarrow{Z} \text{Id} & \text{if } v = e \\
\text{Up}(e,x) \circ \text{Down}(v,z) & \text{otherwise}
\end{cases}
\]

Down(v,z)(1) = T

Diagram:

- a \xrightarrow{1} Rename(b)
- Rename(b) \xrightarrow{} Up(c,3)
- Up(c,3) \xrightarrow{} Down(d,2)
- Down(d,2) \xrightarrow{} d
- d \xrightarrow{4} 1
Example:
Swap the values of two variables v, w

\[
\text{Swap}(v, w)(e, x) = \begin{cases} 
\text{Rename}(v) \circ \text{Down}(w, x) & \text{if } v = e \\
\text{Rename}(w) \circ \text{Down}(v, x) & \text{if } w = e \\
e \xrightarrow{\Delta} \text{Swap}(v, w) & \text{otherwise}
\end{cases}
\]

\[
\text{Swap}(v, w)(1) = 1
\]

\[
\text{Rename}(v)(e, x) = v \xrightarrow{\Delta} \text{Id}
\]

\[
\text{Rename}(v)(1) = T
\]

\[
\text{Up}(v, z)(e, x) = e \xrightarrow{\Delta} v \xrightarrow{\Delta} \text{Id}
\]

\[
\text{Up}(v, z)(1) = T
\]

\[
\text{Down}(v, z)(e, x) = \begin{cases} 
v \xrightarrow{\Delta} v \xrightarrow{\Delta} \text{Id} & \text{if } v = e \\
\text{Up}(e, x) \circ \text{Down}(v, z) & \text{otherwise}
\end{cases}
\]

\[
\text{Down}(v, z)(1) = T
\]

Example:
Swap the values of two variables v, w
Example: Swap the values of two variables v, w

\[
\text{Swap}(v,w)(e,x) = \begin{cases} 
\text{Rename}(v) \circ \text{Down}(w,x) & \text{if } v=e \\
\text{Rename}(w) \circ \text{Down}(v,x) & \text{if } w=e \\
e \xrightarrow{X} \text{Swap}(v,w) & \text{otherwise}
\end{cases}
\]

\[
\text{Swap}(v,w)(1) = 1
\]

\[
\text{Rename}(v)(e,x) = v \xrightarrow{X} \text{Id}
\]

\[
\text{Rename}(v)(1) = T
\]

\[
\text{Up}(v,z)(e,x) = e \xrightarrow{X} v \xrightarrow{Z} \text{Id}
\]

\[
\text{Up}(v,z)(1) = T
\]

\[
\text{Down}(v,z)(e,x) = \begin{cases} 
v \xrightarrow{X} v \xrightarrow{Z} \text{Id} & \text{if } v=e \\
\text{Up}(e,x) \circ \text{Down}(v,z) & \text{otherwise}
\end{cases}
\]

\[
\text{Down}(v,z)(1) = T
\]
Example:
Swap the values of two variables $v, w$

\[
\text{Swap}(v,w)(e,x) =
\begin{cases}
\text{Rename}(v) \circ \text{Down}(w,x) & \text{if } v = e \\
\text{Rename}(w) \circ \text{Down}(v,x) & \text{if } w = e \\
e \xrightarrow{X} \text{Swap}(v,w) & \text{otherwise}
\end{cases}
\]

\[
\text{Swap}(v,w)(1) = 1
\]

\[
\text{Rename}(v)(e,x) = v \xrightarrow{X} \text{Id}
\]
\[
\text{Rename}(v)(1) = T
\]

\[
\text{Up}(v,z)(e,x) = e \xrightarrow{X} v \xrightarrow{Z} \text{Id}
\]
\[
\text{Up}(v,z)(1) = T
\]

\[
\text{Down}(v,z)(e,x) =
\begin{cases}
v \xrightarrow{X} v \xrightarrow{Z} \text{Id} & \text{if } v = e \\
\text{Up}(e,x) \circ \text{Down}(v,z) & \text{otherwise}
\end{cases}
\]
\[
\text{Down}(v,z)(1) = T
\]
Study: “Compute the reachability set”

Problem

- Encoding state set as DDD
- Encoding transition relation as homomorphism

DDD = constant depth
TR = local operations

DDD = dynamic depth
TR = local operations
Technique: “As MDD”

Encoding state set as DDD

- One variable per place
- Define a total order on places

Encoding transition relation as homomorphism

- One inductive homomorphism \( \text{TestSet}(p, pre, post) \)

\[
\text{TestSet}(p, pre, post)(e, x) =
\begin{cases}
  p \xrightarrow{x + \text{post} - \text{pre}} \text{Id} & \text{if } p = e, x \geq \text{pre} \\
  0 & \text{if } p = e, x < \text{pre} \\
  e \xrightarrow{x} \text{TestSet}(p, pre, post) & \text{otherwise}
\end{cases}
\]

\( \text{TestSet}(p, pre, post)(1) = T \)

- Relation for one transition \( R(t) = \prod \text{TestSet operators} \)
- Transition for Petri net = \( \text{Id} + \sum R(t) \) or \( \prod (\text{Id} + R(t)) \)
Inhibitor arcs, Capacity places, Reset arcs

• as ordinary Petri nets

Self Modifying Nets

• DDD as ordinary Petri nets
• 6 new inductive homomorphisms
  — applying \( m(p) = m(p) + m(q) - m(r) \)
  — strongly depend on the order between \( p, q \) and \( r \)

*The 6 new homomorphisms are designed as the “swap operator”*
*(See the paper ICATPN'02, Couvreur et al. for details)*
Encoding state set as DDD

- One variable per place
- One variable per queue but one occurrence per message
- Define a total order on places and queues

|Reach| = 3 states
Encoding state set as DDD

- One variable per place
- One variable per queue but one occurrence per message
- Define a total order on places and queues

|Reach| = 3 states
Encoding state set as DDD

- One variable per place
- One variable per queue but one occurrence per message
- Define a total order on places and queues

\[ |\text{Reach}| = 3 \text{ states} \]
Encoding transition relation as homomorphism

- Two inductive homomorphisms $\text{Send}$, $\text{Rec}$

$\text{Send} = \text{replace } f \xrightarrow{\#} a \text{ by } f \xrightarrow{a} f \xrightarrow{\#} \text{Id}$

$$\text{Send}(f,a)(e,x) = \begin{cases} f \xrightarrow{a} f \xrightarrow{\#} \text{Id} & \text{if } e=f, x=\# \\ e \xrightarrow{x} \text{Send}(f,a) & \text{otherwise} \end{cases}$$

$\text{Send}(f,a)(1) = T$

$\text{Receive} = \text{remove the occurrence of } f \xrightarrow{a}$

$$\text{Rec}(f,a)(e,x) = \begin{cases} \text{Id} & \text{if } e=f, x=a \\ 0 & \text{if } e=f, x\neq a \\ e \xrightarrow{x} \text{Rec}(f,a) & \text{otherwise} \end{cases}$$

$\text{Rec}(f,a)(1) = T$

*Lossy channels are treated using other homomorphisms*
## Experimentation

*Values from ICATPN'02*

| Model                          | N  | Reached     | |DDD| | No sharing          | time |
|-------------------------------|----|-------------|-----|-----|---------------------|------|
| **Philosopher**                |    |             |     |     |                     |      |
| Safe net                      | 5  | 1364        | 127 | 11108| 0.12                |      |
|                               | 10 | 1.86*10⁶    | 267 | 1.52 *10⁷ | 0.68               |      |
|                               | 50 | 2.23*10³¹   | 1387| 1.82 *10³² | 24.48             |      |
| **Fms**                       |    |             |     |     |                     |      |
| Ordinary net                  | 5  | 2.89*10⁶    | 225 | 9.97*10⁶   | 0.76               |      |
|                               | 10 | 2.5*10⁹     | 580 | 8.02*10⁹   | 4.05               |      |
|                               | 20 | 6.03*10¹²   | 1740| 1.87*10¹³  | 26.09              |      |
| **Alternate bit**             |    |             |     |     |                     |      |
| Inhibitor, reset, capacity    | 5  | 14688       | 84  | 43804| 1.89               |      |
|                               | 10 | 170368      | 84  | 480394| 11.21              |      |
|                               | 20 | 2.23*10⁶    | 84  | 6.29*10⁶  | 94.66              |      |
| **Preemptive writer**         |    |             |     |     |                     |      |
| Self modifying net            | 5x5| 873         | 120 | 6440 | 0.62               |      |
|                               | 10x10| 1.94*10⁶ | 485 | 1.46*10⁷ | 18.35             |      |
|                               | 15x15| 4.95*10⁹ | 1100| 3.77*10¹⁰ | 137.5             |      |
| **Alternate bit**             |    |             |     |     |                     |      |
| Lossy Queuing net             | 20 | 21105       | 284 | 63943| 1.57               |      |
|                               | 50 | 280755      | 644 | 845263| 8.46               |      |
|                               | 100| 2.12*10⁶    | 1244| 6.37*10⁶ | 33.4              |      |
Concluding remarks

• DDD are very similar to MDD
  • except for variable length paths:
    • useful in some contexts (FIFO...)
    • forbids (level,index) access

• Homomorphisms are the real difference
  • operations defined independently of the values they work with:
    • \( x \leftarrow x + 1 \)
    • instead of \( 0 \leftarrow 1, 1 \leftarrow 2, \ldots \)
  • compositional (algebraic) framework
  • good expressivity
  • highly extensible/flexible
  • transitive closure operator allows saturation type algorithms
The Saturation Algorithm for Decision diagrams
Algorithm 1: Four variants of a transitive closure loop.

Data: \{Hom\} $T$: the set of transitions encoded as $h_{Trans}$ homomorphisms
$S \ m_0$: initial state encoded as $r(M)$ SDD
$S \ todo$: new states to explore
$S \ reach$: reachable states

\hspace{1cm} \begin{align*}
a) \ \text{Explicit reachability style} & \quad b) \ \text{Standard symbolic BFS loop} \\
\text{begin} & \quad \text{begin} \\
\quad todo := m_0 \quad & \quad todo := m_0 \\
\quad reach := m_0 \quad & \quad reach := 0 \\
\quad \text{while } todo \neq 0 \text{ do} \quad & \quad \text{while } todo \neq reach \text{ do} \\
\quad \quad & \quad \quad \text{reach} := todo \\
\quad \quad & \quad \quad todo := todo + T(todo) \equiv (T + Id)(todo) \\
\quad \quad \text{end} & \quad \text{end} \\
\end{align*}

\hspace{1cm} \begin{align*}
c) \ \text{Chaining loop} & \quad d) \ \text{Saturation enabled} \\
\text{begin} & \quad \text{begin} \\
\quad todo := m_0 \quad & \quad reach := (T + Id)^*(m_0) \\
\quad reach := 0 \quad \quad \text{end} \\
\quad \text{while } todo \neq reach \text{ do} \quad \quad \text{end} \\
\quad \quad reach := todo \\
\quad \quad \text{for } t \in T \text{ do} \\
\quad \quad \quad todo := (t + Id)(todo) \\
\quad \quad \text{end} & \\
\end{align*}
Saturation

- Model-checking using decision diagrams => (nested) transitive closures over the transition relation
- Optimizing complexity of this operation critical to efficiency
- [BCM'92] based on BFS style iterations, n iterations required where n is depth of “deepest” state
- [Roig'95] Chaining may converge faster, based on clusters of transitions, no longer strict BFS
- [Ciardo'01] Saturation is empirically 1 to 3 orders of magnitude better
Saturation vs BFS

- Saturation algorithm: [Ciardo et al. TACAS'01]
  - Fire transitions from the leaves (terminals) up to root
  - Go to ancestor of a node iff. The current node is saturated: all events that only affect this variable and variables below it have been fired until a fixpoint is reached
  - Each time a node is affected by an event, resaturate it.
- Not BFS anymore, firing order of events follows data structure
  - Huge reduction of time and space complexity
  - Good tackling of intermediate peak size effect
- However:
  - Definition of saturation algorithm is complex
  - Cannot be implemented directly with public API of DD libraries

Our contribution: **Automatic saturation**
The transitive closure or *fixpoint* noted * is a unary operator

Evaluated by $h^*(d)$:
- repeat: $d \leftarrow h(d)$
- until: $d = h(d)$

Evaluation may not terminate
- depends on the homomorphism
- if it does, evaluation described as finite composition:
  - $h^* (d) = h \circ h \circ ... h (d)$
  - Thus $h^*$ is a homomorphism

To cumulate states, use of a common construction:
- $(h + id)^*$

Allows to implement a leaf to root saturation strategy
Fixpoint : an example

\[
\text{Seek}(h,v)(e,x) = \begin{cases} 
  h^* \circ e \overset{x}{\to} \text{Id} & \text{if } v=e \\
  e \overset{x}{\to} \text{Seek}(h,v) & \text{otherwise}
\end{cases}
\]

\[
\text{Seek}(h,v)(1) = T
\]

\[
\text{Max}(z)(e,x) = \begin{cases} 
  e \overset{x}{\to} \text{Id} & \text{if } x<z \\
  0 & \text{otherwise}
\end{cases}
\]

\[
\text{Max}(z)(1) = T
\]

\[
h = \text{Max}(3) \circ \text{Inc}(d) \quad // \quad \text{Increment } d \text{ up to } 2
\]
Fixpoint: an example

Seek(h,v)(e,x) = \begin{cases} h^* \circ e \xrightarrow{x} \text{Id} & \text{if } v = e \\ e \xrightarrow{x} \text{Seek}(h,v) & \text{otherwise} \end{cases}

Seek(h,v)(1) = T

Max(z)(e,x) = \begin{cases} e \xrightarrow{x} \text{Id} & \text{if } x < z \\ 0 & \text{otherwise} \end{cases}

Max(z)(1) = T

h = \text{Max}(3) \circ \text{Inc}(d) \quad \text{// Increment } d \text{ up to 2}

A single traversal of these nodes
Fixpoint : an example

Seek(h,v)(e,x) = \[
\begin{cases}
  h^* \circ e \xrightarrow{x} \text{Id} & \text{if } v = e \\
  e \xrightarrow{x} \text{Seek}(h,v) & \text{otherwise}
\end{cases}
\]

Seek(h,v)(1) = T

Max(z)(e,x) = \[
\begin{cases}
  e \xrightarrow{x} \text{Id} & \text{if } x < z \\
  0 & \text{otherwise}
\end{cases}
\]

Max(z)(1) = T

h = Max(3) \circ \text{Inc}(d)  \quad \text{// Increment } d \text{ up to 2}
Fixpoint : an example

Seek(h,v)(e,x) = \[\begin{cases} h^* \circ e & \rightarrow \text{Id} \\ e & \xrightarrow{x} \text{Seek}(h,v) \end{cases}\] if \(v = e\)
otherwise

Seek(h,v)(1) = T

Max(z)(e,x) = \[\begin{cases} e & \xrightarrow{x} \text{Id} \\ 0 \end{cases}\] if \(x < z\)
otherwise

Max(z)(1) = T

\[h = \text{Max}(3) \circ \text{Inc}(d)\] // Increment \(d\) up to 2
Fixpoint : an example

\[ \text{Seek}(h,v)(e,x) = \begin{cases} h^* \circ e \xrightarrow{x} \text{Id} & \text{if } v = e \\ e \xrightarrow{x} \text{Seek}(h,v) & \text{otherwise} \end{cases} \]

\[ \text{Seek}(h,v)(1) = 1 \]

\[ \text{Max}(z)(e,x) = \begin{cases} e \xrightarrow{x} \text{Id} & \text{if } x < z \\ 0 & \text{otherwise} \end{cases} \]

\[ \text{Max}(z)(1) = 1 \]

\[ h = \text{Max}(3) \circ \text{Inc}(d) \quad \text{// Increment } d \text{ up to 2} \]
Fixpoint : an example

Seek(h,v)(e,x) = \begin{cases} 
  h^* \circ e \xrightarrow{x} \text{Id} & \text{if } v=e \\
  e \xrightarrow{x} \text{Seek}(h,v) & \text{otherwise}
\end{cases}

Seek(h,v)(1) = T

Max(z)(e,x) = \begin{cases} 
  e \xrightarrow{x} \text{Id} & \text{if } x<z \\
  0 & \text{otherwise}
\end{cases}

Max(z)(1) = T

h = \text{Max}(3) \circ \text{Inc}(d) \quad \text{/// Increment } d \text{ up to } 2
Fixpoint conclusions

- Transitive closure or fixpoint allows:
  - single traversal of the top of the tree
  - less intermediate nodes
Fixpoint conclusions

- Transitive closure or fixpoint allows:
  - single traversal of the top of the tree => cost of + and h
  - less intermediate nodes

Useless intermediate nodes!!
Saturation effect

- Nested transitive closure or fixpoint = saturation allows:
  - single traversal of the top of the tree => cost of + and h
  - less intermediate nodes

Useless intermediate nodes !!
Performance measures: effect of saturation

- Transitive closure allows more efficiency
- Manual Saturation "à la Ciardo" (Tacas'01 and '03) using * operator
- Organize events by highest variable affected

<table>
<thead>
<tr>
<th>Model</th>
<th>N</th>
<th>Nb. States</th>
<th>final nodes</th>
<th>total nodes</th>
<th>time (s)</th>
<th>total nodes</th>
<th>time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Dining Philosophers</td>
<td>50</td>
<td>2.23e+31</td>
<td>1387</td>
<td>13123</td>
<td>11.6</td>
<td>10739</td>
<td>0.09</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>4.97e+62</td>
<td>2787</td>
<td>26823</td>
<td>54.19</td>
<td>21689</td>
<td>0.18</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>2.47e+125</td>
<td>5587</td>
<td>54223</td>
<td>234</td>
<td>43589</td>
<td>0.39</td>
</tr>
<tr>
<td></td>
<td>1000</td>
<td>9.18e+626</td>
<td>27987</td>
<td></td>
<td></td>
<td>218789</td>
<td>2.1</td>
</tr>
<tr>
<td>Slotted Ring Protocol</td>
<td>10</td>
<td>8.29e+09</td>
<td>1281</td>
<td>35898</td>
<td>83.07</td>
<td>45970</td>
<td>0.8</td>
</tr>
<tr>
<td></td>
<td>15</td>
<td>1.46e+15</td>
<td>2780</td>
<td>118054</td>
<td>595</td>
<td>132126</td>
<td>2.26</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>1.72e+52</td>
<td>29401</td>
<td></td>
<td></td>
<td>3.58e+06</td>
<td>61.58</td>
</tr>
<tr>
<td>Flexible Manufacturing System</td>
<td>10</td>
<td>2.50e+09</td>
<td>580</td>
<td>8604</td>
<td>2.06</td>
<td>11202</td>
<td>0.17</td>
</tr>
<tr>
<td></td>
<td>25</td>
<td>8.54e+13</td>
<td>2545</td>
<td>50489</td>
<td>28.75</td>
<td>85962</td>
<td>1.58</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>4.24e+17</td>
<td>8820</td>
<td>231464</td>
<td>240.4</td>
<td>490062</td>
<td>9.78</td>
</tr>
<tr>
<td></td>
<td>80</td>
<td>1.58e+20</td>
<td>21300</td>
<td></td>
<td></td>
<td>1.72e+06</td>
<td>37.06</td>
</tr>
<tr>
<td>Kanban</td>
<td>10</td>
<td>1.01e+09</td>
<td>257</td>
<td>26862</td>
<td>20.47</td>
<td>5837</td>
<td>0.06</td>
</tr>
<tr>
<td></td>
<td>50</td>
<td>1.04e+16</td>
<td>3217</td>
<td></td>
<td></td>
<td>209117</td>
<td>3.96</td>
</tr>
<tr>
<td></td>
<td>100</td>
<td>1.73e+19</td>
<td>11417</td>
<td></td>
<td></td>
<td>1.32e+06</td>
<td>28.09</td>
</tr>
<tr>
<td></td>
<td>200</td>
<td>3.17e+22</td>
<td>42817</td>
<td></td>
<td></td>
<td>9.23e+06</td>
<td>238.95</td>
</tr>
</tbody>
</table>
Hierarchical Set Decision Diagrams & Automatic Saturation
Alexandre Hamez, Yann Thierry-Mieg, Fabrice Kordon
June 2008 - ICATPN’08

Alexandre Hamez, Yann Thierry-Mieg, Fabrice Kordon
June 2008 - ICATPN’08
LIP6 - LRDE
Xi’an, China
Hierarchical Set Decision Diagram (SDD)

- Limits of DDD reached, need for structure
- Idea: hierarchy
  - Label the arcs with a SET = Set Decision Diagram

With hierarchy

- Increases sharing
  - Memory gain
  - Time gain
    - cache
    - traversals

DDD or SDD = referenced values
• More Formally

\[ E : \text{a set of variables} \]
\[ e \in E, \ \text{Dom}(e) : \text{its (possibly infinite) domain} \]
\[ d \text{ is inductively defined as an SDD iff.} \]

- \[ d \in \{0,1\} \] or
- \[ d = <e, \alpha> \]

- Given a partition \( \pi \) of \( \text{Dom}(e) \)
- \[ \alpha : \pi \mapsto \text{SDD}, \ s.t. \ \forall i,j, i \neq j \Rightarrow \alpha(a_i) \neq \alpha(a_j) \]

Arcs to 0 and arcs labeled by \( \emptyset \) not represented

Fusing Arcs

Splitting arcs
Set Decision Diagrams: A compositional Model

- SDD arcs may be labeled by DDD
  - Hierarchical Structure
  - Adapted to composition
    - captures the similarity of repeated modules

The similarity of behavior of the philosophers is captured:
8*(1 Philosopher)
State space, 4 philosophers (DDD)

With SDD, the state of one philosopher is referenced.
Model "slotted ring", 5 participants, 53856 states

SDD ...
18 nodes

Peak at 1000 nodes
0.14 secondes

...+DDD
32 nodes
Kanban example, low parameter value (5)

Explosion of arcs per node as states per component increases

MDD (Smart)
• SDD well adapted to composition
  • The modules give structuration
    • SDD put in relation sets of states
  • On the fly computation of synchronized products
  • Gain of an order of magnitude over DDD
Definition

\[ \Phi : \text{SDD} \rightarrow \text{SDD} \]

\[ \forall d_1, d_2 \quad \begin{cases} 
\Phi(0) = 0 \\
\Phi(d_1) + \Phi(d_2) = \Phi(d_1 + d_2) 
\end{cases} \]

Examples, hard coded in the library

Id, Id + d, Id * d, Id \ d, Id . d, d.Id

Proposition

If \( \Phi_1, \Phi_2 \) are homomorphisms then

\( \Phi_1 + \Phi_2 \) and \( \Phi_1 \circ \Phi_2 \) are homomorphisms
Inductive Homomorphism

• Used to define user operations
  • Flexible, and powerful
  • Benefits from a cache

• $\Phi$ is inductively defined by:
  • $\Phi(1) \in SDD$: constant terminal case
  • $\Phi(e,x) \in Hom$: $e$ in $E$, $x \subset \text{Dom}(e)$: evaluation for an arbitrary SDD arc returns a homomorphism to apply on successor node
Skip predicate (NEW)

- \textbf{Skip(e)} expresses local invariance: \textit{Skip is true }\Rightarrow\textit{ the variable is neither read nor written}

\[
\phi \left( \begin{array}{c}
\vdots & a_i & \vdots \\
\downarrow & \downarrow & \downarrow \\
d_i & \text{e} & \text{e}
\end{array} \right) \rightarrow \begin{array}{c}
\vdots & a_i & \vdots \\
\downarrow & \downarrow & \downarrow \\
d_i & \phi(d_i)
\end{array}
\]

- Extends to composition: \textit{f+g and } f \circ g \textit{ skip a variable } e \textit{ iff. both operands skip(e)}

- \textit{Minimal structural information} about user operations that allows to enable saturation automatically
Fixpoint operator : *

- Built-in operator for transitive closure
- \( \Phi^* (d) = \Phi^n (d) \), where \( n \) is the smallest integer such that
  \[ \Phi^{n+1} (d) = \Phi^n (d) \]
- May not terminate (\( n \) infinite)
- Most often used as an accumulator:
  \( (\Phi + \text{Id})^* \)
- Transitive closure naturally expressed as:
  \( (t_1 + t_2 + \ldots + t_n + \text{Id})^* \)

Allows library to automatically enable saturation
Homomorphisms for a Petri Net

- **Pre arc homomorphism**
  
  \[ h^-(p, v)(e, x) = \]
  
  \[ \begin{cases} 
  e \xrightarrow{x-v} \text{Id} & \text{if } x \geq v \\
  0 & \text{if } x < v 
  \end{cases} \]

  \[ h^- \cdot \text{Skip}(e) = (e \neq p) \]

  \[ h^-(p, v)(1) = 0 \]

- **Post arc homomorphism**
  
  \[ h^+(p, v)(e, x) = e \xrightarrow{x+v} \text{Id} \]

  \[ h^+ \cdot \text{Skip}(e) = (e \neq p) \]

  \[ h^+(p, v)(1) = 0 \]

- h- returns terminal 0 to prune path if precondition not met
- Skip on e ≠ p: only one variable (place) is affected by the arc

- For a full transition, compose Post after Pre, e.g.

\[ h_{Trans}(hungry) = h^+(WaitL, 1) \circ h^+(WaitR, 1) \circ h^-(Idle, 1) \]
• Built-in "local" homomorphism allows to target arc value(s) of a given variable

\[
\begin{align*}
local(h, var)(e, x) &= e \xrightarrow{h(x)} Id \\
local(h, var).\text{Skip}(e) &= (e \neq var) \\
local(h, var)(1) &= 0
\end{align*}
\]
DDD sharing & SDD sharing (reminder)

State space, 4 philosophers (DDD)

With SDD, the state of one philosopher is referenced.
Homomorphisms for Labeled Petri net

- Built-in “local” homomorphism allows to target arc value(s) of a given variable

\[
\text{local}(h, \text{var})(e, x) = e \xrightarrow{h(x)} \text{Id}
\]

\[
\text{local}(h, \text{var}).\text{Skip}(e) = (e \neq \text{var})
\]

\[
\text{local}(h, \text{var})(1) = 0
\]

- Full transition relation for a synchronization is built as a composition of local operations: e.g. philo P0 finishes eating:

\[
= \text{local}(h_{\text{Trans}}(\text{eat}), m\((P_0)\))
\]

\[
\circ \text{local}(h_{\text{Trans}}(\text{putFork}), m\((P_1)\))
\]

\[
= \text{local}(h^+(\text{Idle}, 1) \circ h^+(\text{Fork}, 1) \circ h^-(\text{HasL}, 1) \circ h^-(\text{HasR}, 1), m\((P_0)\))
\]

\[
\circ \text{local}(h^+(\text{Fork}, 1), m\((P_1)\))
\]
Rewriting rules for homomorphisms

• For Union: \( H = \sum g_1 + \ldots + g_n + \sum f_1 + \ldots + f_m \) such that on current variable \( g \) terms do not skip and \( f \) terms skip.

\[
H = \begin{pmatrix}
  \ldots & a_i & \ldots \\
  d_i & & \\
\end{pmatrix} \quad \rightarrow \quad G = \begin{pmatrix}
  \ldots & a_i & \ldots \\
  d_i & & \\
\end{pmatrix} + F_{(d_i)}
\]

• No \textit{a priori} variable order => Partition of operands of union is cached
Effect of skip on \((H + Id)^*\)

- **Case of interest:** transitive closure of a set of transitions + Id

\[
\begin{align*}
(H+Id)^* & \rightarrow (G+Id)^* \\
& \rightarrow (F+Id)^* \\
& \rightarrow (L+Id)^* \\
& \rightarrow (F+Id)^* \\
\end{align*}
\]

- **Additional rules are defined for “local” construction (see proceedings)**
- **Essentially,** \((\text{Local}(h,v) + id)^* = \text{Local}( (h+id)^*, v )\)
### Performances

<table>
<thead>
<tr>
<th>Model Size</th>
<th>States #</th>
<th>DDD</th>
<th>SDD</th>
<th><strong>Final #</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>DDD</td>
<td>SDD</td>
<td>Hierarchical Chaining Loop</td>
</tr>
<tr>
<td>LOTOS Specification</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>9.8e+21</td>
<td>–</td>
<td>1085</td>
<td>–</td>
<td>–</td>
</tr>
<tr>
<td>Dining Philosophers</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>4.9e+62</td>
<td>2792</td>
<td>419</td>
<td>1.9</td>
</tr>
<tr>
<td>200</td>
<td>2.5e+125</td>
<td>5589</td>
<td>819</td>
<td>7.9</td>
</tr>
<tr>
<td>1000</td>
<td>9.2e+626</td>
<td>27989</td>
<td>4019</td>
<td>–</td>
</tr>
<tr>
<td>4000</td>
<td>7e+2507</td>
<td>–</td>
<td>16019</td>
<td>–</td>
</tr>
<tr>
<td>Slotted Ring Protocol</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>8.3e+09</td>
<td>1283</td>
<td>105</td>
<td>1.1</td>
</tr>
<tr>
<td>50</td>
<td>1.7e+52</td>
<td>29403</td>
<td>1345</td>
<td>–</td>
</tr>
<tr>
<td>100</td>
<td>2.6e+105</td>
<td>–</td>
<td>5145</td>
<td>–</td>
</tr>
<tr>
<td>150</td>
<td>4.5e+158</td>
<td>–</td>
<td>11445</td>
<td>–</td>
</tr>
</tbody>
</table>
To finish:
Compositional nesting & hierarchy

- SDD arcs may reference SDD
  - Hierarchical structure
  - Arbitrary depth
- Exemple Philosophers:

\[ 2^3 = 8 \] philosophes:
3 levels of depth
+ representation of a philosopher
Sharing at every level
Philosophers & Hierarchy: Potential of SDD

Y. Thierry-Mieg – Octobre 2008

<table>
<thead>
<tr>
<th>Nb. Philosophers</th>
<th>States</th>
<th>Time (s)</th>
<th>Final</th>
<th>Peak</th>
</tr>
</thead>
<tbody>
<tr>
<td>$2^{10}$</td>
<td>$1.02337e+642$</td>
<td>0.0</td>
<td>83</td>
<td>717</td>
</tr>
<tr>
<td>$2^{31}$</td>
<td>$1.63233e+1346392620$</td>
<td>0.02</td>
<td>251</td>
<td>2250</td>
</tr>
<tr>
<td>$2^{1000}$</td>
<td>$N/A$</td>
<td>0.81</td>
<td>8003</td>
<td>72987</td>
</tr>
<tr>
<td>$2^{10000}$</td>
<td>$N/A$</td>
<td>9.85</td>
<td>80003</td>
<td>729987</td>
</tr>
<tr>
<td>$2^{20000}$</td>
<td>$N/A$</td>
<td>20.61</td>
<td>160003</td>
<td>1459987</td>
</tr>
</tbody>
</table>

Final | Peak |
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>SDD</td>
<td>DDD</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
<tr>
<td>SDD</td>
<td>DDD</td>
</tr>
<tr>
<td>-------</td>
<td>------</td>
</tr>
</tbody>
</table>

Philosophers States

P1

(P1)

P7

(P7)

P8

(P8)
Conclusion

• Latest evolution of decision diagrams: SDD
  • Suitable for very large systems
  • Well suited to hierarchical/compositional specifications

• Transparent and efficient transitive closure
  • User defined homomorphisms
  • Automatic saturation: generalize [Ciardo’01] with definition independent of a given formalism

• Recursive folding for logarithmic complexity on some examples

• SDD and DDD distributed as an open-source LGPL C++ library: http://www.lip6.fr/ddd
Instantiable Transition Systems to Exploit SDD
- Encouraging results obtained with model instance philosophers
- Need for generalization
  - A framework to define structured models
  - Notion of type and instance
  - Composite types contain nested instances
  - Transition relation defined as:
    - Local events to an instance
    - Synchronizations of transition labels of contained instances
Definition 2 (ITS Concepts). An ITS type must provide a tuple
type $= \langle S, \text{InitStates}, T, \text{Locals}, \text{Succ} \rangle$:

- $S$ is a set of states;
- $\text{InitStates} \subseteq S$ is a finite subset of designated initial states;
- $T$ is a finite set of public transition labels;
- $\text{Locals} : S \rightarrow 2^S$ is the local successors function.
- $\text{Succ} : S \times \text{Bag}(T) \rightarrow 2^S$ is the transition function satisfying $\forall s \in S, \text{Succ}(s, \emptyset) = \{s\}$.

Let Types denote a set of ITS types. An ITS instance $i$ is defined by its ITS type, noted $\text{type}(i) \in \text{Types}$.

SDD State Encoding,
Locals() and $\text{Succ}(t_1+\ldots+t_n)$ as homomorphisms
An example

Buffer

put
get
empty

write
read

Process

get_token
give_token

active
passive

Pu
Ty
Ini
A composite type

initial = {b->empty, p1->active, p2->passive}
An elementary type (LTS)

Process

- ask → 1 → give_token
- read → 3 → write → 5 → get_token
- get_token → 4 → read → 6 → write

active = 0
passive = 1

private={ask}
public={get_token,give_token,read,write}
XOR non deterministic Synchronizations

```
active = {p1->active, p2->passive}
passive = {p1->passive, p2->passive}
```
Closing a ring

```
initial = {p->active}
```
A full system

RobinSystem

\[
\text{initial} = \{ b \rightarrow \text{empty}, p \rightarrow \text{initial} \}
\]
Experimentations (time)

ITS Time (s)

Smart Time (s)
Conclusions ITS

- Still work in progress...
- Provides a (general) way of capturing patterns of similar behaviors
- Experiments with hierarchy show that most models of the benchmark can be encoded more efficiently with SDD
- An easy way to profit from ITS: define your own (elementary) type(s)
  - SDD state encoding
  - Homomorphism transition encoding
- Extensions of composite type also possible
  - Reset transitions, priorities...