LTSmin Tool Architecture (1)

- Specification Languages
  - mCRL2
  - Promela
  - DVE
  - UPPAAL

- Reachability Tools
  - Distributed
  - Multi-core
  - Symbolic

- PINS

- Functionality
  - On-the-fly detection of errors: deadlocks, actions, invariant violations
  - On-the-fly LTL model checking for liveness (Nested DFS)
  - Symbolic model checker for CTL*, full $\mu$-calculus
  - State space generation, bisimulation minimization, export
  - State and edge labels support timed and stochastic systems
**Functionality**

- On-the-fly detection of errors: deadlocks, actions, invariant violations
- On-the-fly LTL model checking for liveness (**Nested DFS**)
- **Symbolic model checker** for CTL*, full $\mu$-calculus
- State space generation, **bisimulation minimization**, export
- State and edge labels support timed and stochastic systems
Guard-based Partial-Order Reduction in LTSmin

UNIVERSITY OF TWENTE.

Guard-based Partial-Order Reduction in LTSmin

8 july 2013
Partitioned Interface for Next States:

- States are partitioned into vector of $N$ state variables
- The next-state function is partitioned into $M$ transition groups
- Show locality: $N \times M$ dependency matrix (hopefully sparse)
  - indicates which state parts each transition group depends on
Partitioned Interface for Next States:

- States are partitioned into vector of $N$ state variables
- The next-state function is partitioned into $M$ transition groups
- Show locality: $N \times M$ dependency matrix (hopefully sparse)
  - indicates which state parts each transition group depends on

On-the-fly access to the state space via an API:

Three basic functions

- INIT-STATE(): returns the initial state vector
- NEXT-STATE(i,s): successors of state $s$ in transition group $i$
- GET-MATRIX: returns the dependency matrix $D_{M \times N}$
Dependency Matrix: caching and regrouping

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

Guard-based Partial-Order Reduction in LTSmin
8 july 2013 5 / 16
Dependency Matrix: caching and regrouping

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

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

global int z=9;
process p3() {
do  ::= {z>0 \rightarrow z--; x++}
 ::= {z>0 \rightarrow z--; y++}
od }}
Dependency Matrix: caching and regrouping

```latex
\begin{align*}
\text{global int } & x=7; \\
\text{process } & p1() \{ \\
& \text{do} \\
& \quad \{x>0 \rightarrow x--; y++\} \\
& \quad \{x>0 \rightarrow x--; z++\} \\
& \text{od } \}
\end{align*}
```

```latex
\begin{align*}
\text{global int } & y=3; \\
\text{process } & p2() \{ \\
& \text{do} \\
& \quad \{y>0 \rightarrow y--; x++\} \\
& \quad \{y>0 \rightarrow y--; z++\} \\
& \text{od } \}
\end{align*}
```

```latex
\begin{align*}
\text{global int } & z=9; \\
\text{process } & p3() \{ \\
& \text{do} \\
& \quad \{z>0 \rightarrow z--; x++\} \\
& \quad \{z>0 \rightarrow z--; y++\} \\
& \text{od } \}
\end{align*}
```

### Process Matrix

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

### In general:

using \(r/w/+\)
Dependency Matrix: caching and regrouping

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

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

global 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>p1</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>p2</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
<tr>
<td>p3</td>
<td>+</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

In general:
using r/w/+
Dependency Matrix: caching and regrouping

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

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

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

Process Matrix

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

In general:
using r/w/+  

Refined Matrix

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

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

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

cache short transitions
enable symbolic means
### Dependency Matrix: caching and regrouping

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

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

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

**Process Matrix**

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

**Refined Matrix**

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

**Static Regrouping**

<table>
<thead>
<tr>
<th></th>
<th>x</th>
<th>y</th>
<th>z</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1.1,2.1</td>
<td>+</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>p1.2,3.1</td>
<td>+</td>
<td>-</td>
<td>+</td>
</tr>
<tr>
<td>p2.2,3.2</td>
<td>-</td>
<td>+</td>
<td>+</td>
</tr>
</tbody>
</table>

- Less overhead
- Better structure
# Table of Contents

1. Introduction LTSmin
   - LTSmin Tool Architecture
   - PINS Interface

2. Theory
   - Basis: Stubborn Sets
   - Guard Based POR
   - Necessary Disabling Sets

3. Implementation
   - Language Module Extensions
   - Algorithm to find small Stubborn Sets
   - POR and LTL model checking

4. Experiments

5. Conclusion
Partial-Order Reduction

(Godefroid, Valmari)

Main idea of partial-order reduction

- Avoid exploring *all* transition interleavings
- Select *sufficient* subset of enabled transitions
  - don’t destroy *conflicting* transitions
Partial-Order Reduction

(Main idea of partial-order reduction)
- Avoid exploring *all* transition interleavings
- Select *sufficient* subset of enabled transitions
  - don’t destroy conflicting transitions

(Necessary Enabling Sets (NES))
- If transition $\alpha$ is not enabled in state $s$, then
- $NES(\alpha, s)$ is some necessary enabling set
  - it contains a transition from each path to $\alpha$
Partial-Order Reduction

(Godefroid, Valmari)

Main idea of partial-order reduction
- Avoid exploring all transition interleavings
- Select sufficient subset of enabled transitions
  - don’t destroy conflicting transitions

Necessary Enabling Sets (NES)
- If transition \( \alpha \) is not enabled in state \( s \), then
  - \( NES(\alpha, s) \) is some necessary enabling set
    - it contains a transition from each path to \( \alpha \)

Algorithm to compute a Stubborn Set
1. Select an arbitrary enabled transition in \( T_s \)
2. Repeat, for each \( \alpha \in T_s \):
   1. If \( \alpha \) enabled: add all conflicting transitions \( \beta \) to \( T_s \)
   2. If \( \alpha \) disabled: add all transitions in some \( NES(\alpha, s) \) to \( T_s \)
Innovation 1: Guard-centric approach

Atomic transitions: $g_1(\vec{x}) \land \cdots \land g_n(\vec{x}) \rightarrow \vec{x} := t$

Extend \texttt{PINS} with a function to evaluate guards

Define all notions on guards rather than transitions

- guards $x > 0$ and $x < 5$ may be co-enabled \ldots \ldots $MC(g_1, g_2)$
- guards $x = 0$ and $x > 5$ cannot be co-enabled
- guards $pc = 3$ and $pc = 5$ cannot be co-enabled
Innovation 1: Guard-centric approach

Atomic transitions: $g_1(\vec{x}) \land \cdots \land g_n(\vec{x}) \longrightarrow \vec{x} := t$

Extend Pins with a function to evaluate guards

Define all notions on guards rather than transitions

- guards $x > 0$ and $x < 5$ may be co-enabled ...... $MC(g_1, g_2)$
- guards $x = 0$ and $x > 5$ cannot be co-enabled
- guards $pc = 3$ and $pc = 5$ cannot be co-enabled
- How to enable a guard $pc = 3$? ................. $NES(g_1)$
  - Add all transitions that assign $pc := 3$
Innovation 1: Guard-centric approach

Atomic transitions: \( g_1(\vec{x}) \land \cdots \land g_n(\vec{x}) \rightarrow \vec{x} := t \)

Extend \( \text{PINS} \) with a function to evaluate guards

**Define all notions on guards rather than transitions**

- guards \( x > 0 \) and \( x < 5 \) may be co-enabled \( \ldots \ldots MC(g_1, g_2) \)
- guards \( x = 0 \) and \( x > 5 \) cannot be co-enabled
- guards \( pc = 3 \) and \( pc = 5 \) cannot be co-enabled
- How to enable a guard \( pc = 3 \)? \( \ldots \ldots \ldots \ldots NES(g_1) \)
  - Add all transitions that assign \( pc := 3 \)
- An update \( x := 5 \) conflicts with guard \( x + y = z \) \( \ldots \ldots \ldots \ldots DNA \)
- An update \( v := 5 \) doesn’t conflict with guard \( x + y = z \)
- An update \( x := x + 1 \) doesn’t conflict with guard \( x + y > z \)
Innovation 1: Guard-centric approach

Atomic transitions: \( g_1(\vec{x}) \land \cdots \land g_n(\vec{x}) \rightarrow \vec{x} := t \)

Extend \textsc{Pins} with a function to evaluate guards

Define all notions on guards rather than transitions

- guards \( x > 0 \) and \( x < 5 \) may be co-enabled \( \ldots \ldots \MC(g_1, g_2) \)
- guards \( x = 0 \) and \( x > 5 \) cannot be co-enabled
- guards \( pc = 3 \) and \( pc = 5 \) cannot be co-enabled
- How to enable a guard \( pc = 3 \)? \( \ldots \ldots \ldots \ldots \NS(g_1) \)
  - Add all transitions that assign \( pc := 3 \)
- An update \( x := 5 \) conflicts with guard \( x + y = z \) \( \ldots \ldots \DNA \)
- An update \( v := 5 \) doesn’t conflict with guard \( x + y = z \)
- An update \( x := x + 1 \) doesn’t conflict with guard \( x + y > z \)

Program counters or process locations are treated no different than just any other state variable
Innovation 2: Necessary Disabling Sets

Keeping stubborn sets small
- Assume \((t_1, t_7)\) and \((t_6, t_7)\) are conflicting
- Typically, \(NES\) works backwards:
  - Fat stubborn set: \(\{t_1, t_2...5, t_6, t_7\}\)
Innovation 2: Necessary Disabling Sets

Keeping stubborn sets small

- Assume \((t_1, t_7)\) and \((t_6, t_7)\) are conflicting
- Typically, NES works backwards:
  - Fat stubborn set: \(\{t_1, t_2, \ldots, 5, t_6, t_7\}\)
- Note: \(t_1\) and \(t_6\) may not be co-enabled
- Disabling \(t_1\) is necessary to enable \(t_6\):
  - \(\{t_1, t_6, t_7\}\) is a sufficient stubborn set
Innovation 2: Necessary Disabling Sets

Keeping stubborn sets small

- Assume \((t_1, t_7)\) and \((t_6, t_7)\) are conflicting
- Typically, NES works backwards:
  - Fat stubborn set: \(\{t_1, t_2, \ldots, 5, t_6, t_7\}\)
  - Note: \(t_1\) and \(t_6\) may not be co-enabled
- Disabling \(t_1\) is necessary to enable \(t_6\):
  - \(\{t_1, t_6, t_7\}\) is a sufficient stubborn set

Necessary Disabling Sets

- So, how to find an necessary enabling transition for \(\alpha\)?
Innovation 2: Necessary Disabling Sets

Keeping stubborn sets small

- Assume \((t_1, t_7)\) and \((t_6, t_7)\) are conflicting
- Typically, NES works backwards:
  - Fat stubborn set: \(\{t_1, t_2 \ldots 5, t_6, t_7\}\)
- Note: \(t_1\) and \(t_6\) may not be co-enabled
- Disabling \(t_1\) is necessary to enable \(t_6\):
  - \(\{t_1, t_6, t_7\}\) is a sufficient stubborn set

Necessary Disabling Sets

- So, how to find a necessary enabling transition for \(\alpha\)?
- Disable any enabled transition \(\beta\) that is not co-enabled with \(\alpha\)
- \(NDS(\beta, s)\) contains some transition necessary to disable \(\beta\)
Table of Contents

1 Introduction LTSmin
   - LTSmin Tool Architecture
   - PINS Interface

2 Theory
   - Basis: Stubborn Sets
   - Guard Based POR
   - Necessary Disabling Sets

3 Implementation
   - Language Module Extensions
   - Algorithm to find small Stubborn Sets
   - POR and LTL model checking

4 Experiments

5 Conclusion
Language Module Extensions

What every language must provide

- Dependency Matrix for state variables and guards ....... DM
  - distinguish read/write dependencies
- Matrix to report conflicting transitions ................. DNA
Language Module Extensions

What every language must provide

- Dependency Matrix for state variables and guards ...... DM
  - distinguish read/write dependencies
- Matrix to report conflicting transitions .................. DNA

Optional improvements for more reduction

- Necessary Enabling Sets for guards ...................... NES
- Necessary Disabling Sets for guards ..................... NDS
- May-be Co-enabled matrix on guards ..................... MC

All matrices can be approximated by static analysis
A good default can be computed for the optional information
We did extend the language modules for Promela and DVE
Language Module Extensions

What every language must provide

- Dependency Matrix for state variables and guards .... DM
  - distinguish read/write dependencies
- Matrix to report conflicting transitions ................. DNA

Optional improvements for more reduction

- Necessary Enabling Sets for guards ..................... NES
- Necessary Disabling Sets for guards .................... NDS
- May-be Co-enabled matrix on guards ..................... MC

- All matrices can be approximated by static analysis
- A good default can be computed for the optional information
- We did extend the language modules for Promela and DVE
Heuristics for finding Stubborn Sets

Implementation of Stubborn Sets

- **Heuristics** to choose stubborn set with *minimum costs*
  - enabled transitions more expensive than disabled transitions
  - transitions that were selected already come for free
Implementation of Stubborn Sets

- **Heuristics** to choose stubborn set with **minimum costs**
  - enabled transitions more expensive than disabled transitions
  - transitions that were selected already come for free
- This is sufficient for **reachability/deadlock**
  - for the sequential + parallel algorithms
Implementation of Stubborn Sets

- **Heuristics** to choose stubborn set with minimum costs
  - enabled transitions more expensive than disabled transitions
  - transitions that were selected already come for free
- This is sufficient for reachability/deadlock
  - for the sequential + parallel algorithms

Extra implemented provisos (Holzmann, Peled)

- Incorporated extra features in algorithm + language module:
  - Extra: provide visibility information
  - Extra: implemented several cycle provisos
Heuristics for finding Stubborn Sets

Implementation of Stubborn Sets

- Heuristics to choose stubborn set with minimum costs
  - enabled transitions more expensive than disabled transitions
  - transitions that were selected already come for free
- This is sufficient for reachability/deadlock
  - for the sequential + parallel algorithms

Extra implemented provisos (Holzmann, Peled)

- Incorporated extra features in algorithm + language module:
  - Extra: provide visibility information
  - Extra: implemented several cycle provisos
- This is sufficient for LTL model checking
  - only for the sequential algorithms
The Tower of \textsc{Pins} Layers: LTL with POR

Stretching the \textsc{Pins} interface
- Get new transitions \textit{on-the-fly}
  - request from upper layer
  - call-back on each successor
- POR layer needs extra info:
  - visibility from Büchi product
  - cycle-provisor from NDFS

Guard-based Partial-Order Reduction in LTSmin

8 july 2013
The Tower of **PINS** Layers: LTL with POR

**System spec** → **Property \( \varphi \)**

- **Language module**
- **Transition cache**
- **Partial-order reduction**
- **LTL crossproduct**
- **NDFS emptiness check**

**Stretching the PINS interface**

- Get new transitions **on-the-fly**
  - request from upper layer
  - call-back on each successor
- POR layer needs extra info:
  - **visibility** from Büchi product
  - **cycle-proviso** from NDFS

**Refined Proviso’s**

- Cycles: **color proviso**
  - Valmari, Evangelista
- Visibility: **atoms as guards**
  - Reuse en/dis-abling info
  - Dynamic (per state)
Experimental Results
Experimental Results

46 DVE models from BEEM database

- Compare stubborn sets versus ample sets (theory):
  - Reduce more than best possible ample set (Geldenhuys)
  - Heuristics for selecting stubborn sets are very effective
  - Necessary Disabling Sets can help as well
Experimental Results

46 DVE models from BEEM database

- Compare stubborn sets versus ample sets (theory):
  - Reduce more than best possible ample set (Geldenhuys)
  - Heuristics for selecting stubborn sets are very effective
  - Necessary Disabling Sets can help as well

16 Promela models, up to 50M states, 250M transitions

- Compare stubborn sets (LTSmin) with ample sets (SPIN)
  - LTSmin por provides more reduction than Spin por
  - Spin’s partial-order reduction is more efficient in time
  - LTSmin requires less memory (reduction + state compression)
Experimental Results

46 DVE models from BEEM database
- Compare stubborn sets versus ample sets (theory):
  - Reduce more than best possible ample set (Geldenhuys)
  - Heuristics for selecting stubborn sets are very effective
  - Necessary Disabling Sets can help as well

16 Promela models, up to 50M states, 250M transitions
- Compare stubborn sets (LTSmin) with ample sets (SPIN)
  - LTSmin por provides more reduction than Spin por
  - Spin’s partial-order reduction is more efficient in time
  - LTSmin requires less memory (reduction + state compression)

POR combined with LTL model checking
- Guard-based dynamic visibility proviso pays off
- Subtle cycle proviso’s (Valmari, Evangelista) pay off
Guard-based Partial-Order Reduction in LTSmin

Introduction

LTSmin

Theory

POR

Implementation

Experiments

Conclusion
Why join the LTSmin project?

- **Specification Languages**:
  - mCRL2
  - Promela
  - DVE
  - UPPAAL

- **PINS**:
  - Transition caching
  - Variable reordering
  - Transition grouping
  - Partial-order reduction

- **Pins2pins Wrappers**

- **Reachability Tools**:
  - Distributed
  - Multi-core
  - Symbolic
Why join the LTSmin project?

End users: profit without changing modeling language

- probably the best **scalable** model checker up to 48 cores
- economic with **memory** (lossless compression, por reduction)
- supports major modeling languages: **SPIN, UPPAAL, mCRL2**

Spec Lang

Pins2pins Wrappers

PINS

Reachability Tools

- Transition caching
- Variable reordering
- Partial–order reduction

- Distributed
- Multi–core
- Symbolic
Why join the LTSmin project?

End users: profit without changing modeling language
- probably the best **scalable** model checker up to 48 cores
- economic with **memory** (lossless compression, por reduction)
- supports major modeling languages: SPIN, UPPAAL, mCRL2

Developers: build your own HP Domain Specific Model Checker
- easy to link to new language modules through API + matrices
- now provides LTL model checker with partial-order reduction
- provides multi-core, distributed and symbolic algorithms

Reachability Tools
- Distributed
- Multi-core
- Symbolic
Why join the LTSmin project?

**End users: profit without changing modeling language**
- probably the best **scalable** model checker up to 48 cores
- economic with **memory** (lossless compression, por reduction)
- supports major modeling languages: SPIN, UPPAAL, mCRL2

**Developers: build your own HP Domain Specific Model Checker**
- easy to link to new language modules through API + matrices
- now provides LTL model checker with partial-order reduction
- provides multi-core, distributed and symbolic algorithms

**Scientists: prototype, benchmark, compare and combine**
- **symbolic, partial-order reduction, multi-core** in one framework

---

UNIVERSITY OF TWENTE.
Guard-based Partial-Order Reduction in LTSmin
8 july 2013
16 / 16