UNIVERSITY OF TWENTE.

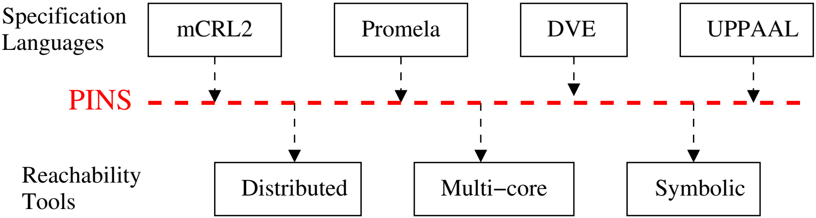
Formal Methods & Tools.

Guard-based Partial-Order Reduction in LTSmin

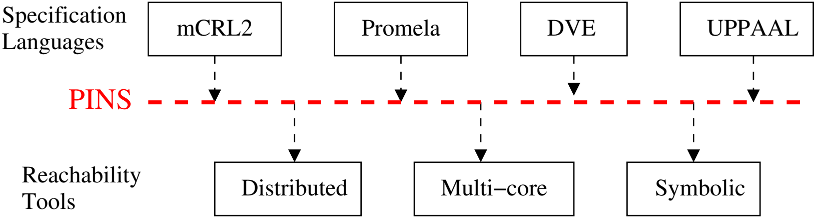
Alfons Laarman, Elwin Pater,
Jaco van de Pol, Michael Weber
8 july 2013

SPIN'13, Stony Brook

LTSmin Tool Architecture (1)



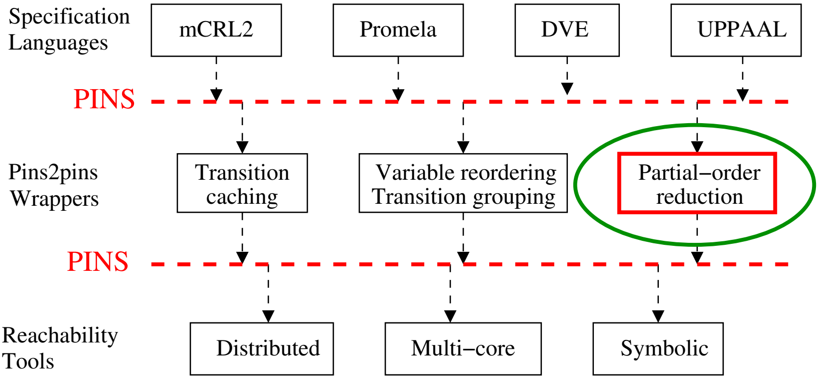
LTSmin Tool Architecture (1)



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 μ-calculus
- State space generation, bisimulation minimization, export
- State and edge labels support timed and stochastic systems

LTSmin Tool Architecture (2)



PINS interface

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

PINS interface

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 }
```

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 }
```

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 }
```

Process Matrix

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

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 }
```

Process Matrix

$$\begin{array}{c c} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} p1 \\ p2 \\ p3 \end{array} & \left[\begin{array}{ccc} + & + & + \\ + & + & + \\ + & + & + \end{array}\right] \end{array}$$

Refined Matrix

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

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 }
```

Process Matrix

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

In general:

using r/w/+

Refined Matrix

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

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

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

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

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

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

cache short transitions

enable symbolic means

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 }
```

Process Matrix

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

In general:
using r/w/+

Refined Matrix

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

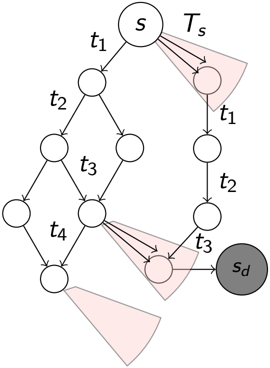
Static Regrouping

$$\begin{array}{cc} & \begin{array}{ccc} x & y & z \end{array} \\ \begin{array}{c} p1.1, 2.1 \\ p1.2, 3.1 \\ p2.2, 3.2 \end{array} & \begin{bmatrix} + & + & - \\ + & - & + \\ - & + & + \end{bmatrix} \end{array}$$

- Less overhead
- Better structure

Table of Contents

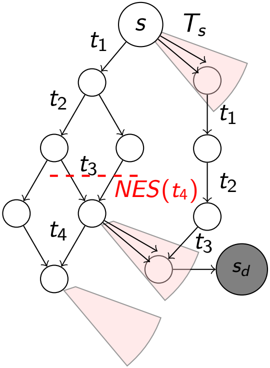
Partial-Order Reduction (Godefroid, Valmari)



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 (Godefroid,Valmari)



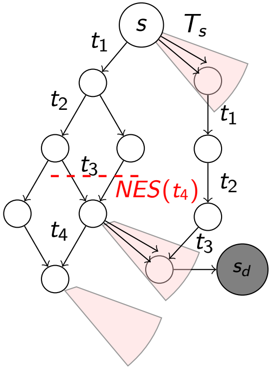
Main idea of partial-order reduction

- Avoid exploring *all* transition interleavings
- Select sufficient subset of enabled transitions
  - don't destroy conflicting transitions

Necessary Enabling Sets (NES)

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

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

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 α enabled: add all conflicting transitions β to T_s
   2. If α disabled: add all transitions in some $NES(\alpha, s)$ to T_s

Innovation 1: Guard-centric approach

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

Innovation 1: Guard-centric approach

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

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 $pc := 3$
- An update $x := 5$ conflicts with guard $x + y = z$ ……. 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

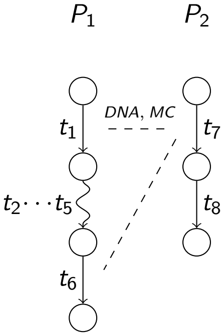
Atomic transitions: $g_1(\vec{x}) \wedge \cdots \wedge g_n(\vec{x}) \longrightarrow \vec{x} := t$
Extend PINS with a function to evaluate guards

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 $pc := 3$
- An update $x := 5$ conflicts with guard $x + y = z$ .... 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

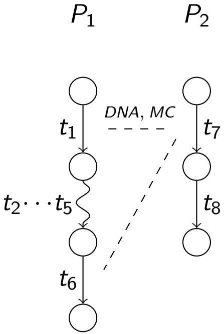
Innovation 2: Necessary Disabling Sets



Keeping stubborn sets small

- Assume (t_1, t_7) and (t_6, t_7) are conflicting
- Typically, *NES* works backwards:
    - Fat stubborn set: $\{t_1, t_{2\ldots5}, t_6, t_7\}$

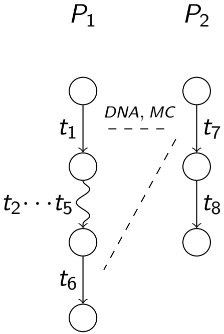
Innovation 2: Necessary Disabling Sets



Keeping stubborn sets small

- Assume (t_1, t_7) and (t_6, t_7) are conflicting
- Typically, *NES* works backwards:
  - Fat stubborn set: $\{t_1, t_{2\ldots5}, t_6, t_7\}$
- 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



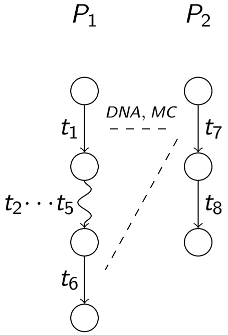
Keeping stubborn sets small

- Assume (t_1, t_7) and (t_6, t_7) are conflicting
- Typically, *NES* works backwards:
  - Fat stubborn set: $\{t_1, t_{2\ldots5}, t_6, t_7\}$
- 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 α?

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: $\{t_1, t_{2\ldots5}, t_6, t_7\}$
- 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 α?
- Disable any enabled transition β that is not co-enabled with α
- $NDS(\beta, s)$ contains some transition necessary to disable β

Table of Contents

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

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

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

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

Heuristics for finding Stubborn Sets

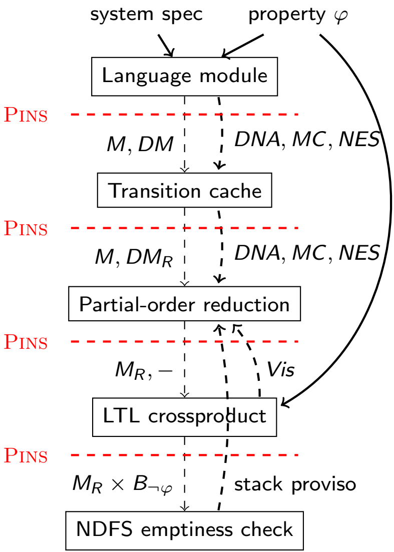
Implementation of Stubborn Sets

- 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

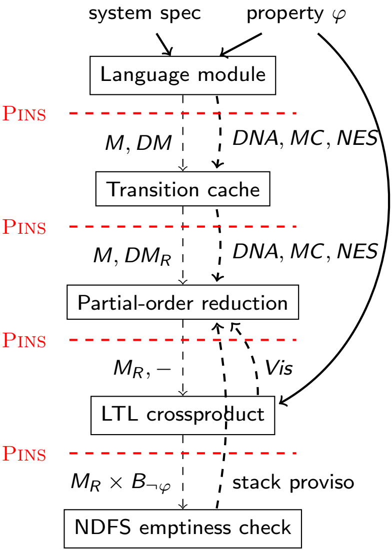
The Tower of PINS Layers: LTL with POR



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

The Tower of PINS Layers: LTL with POR



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

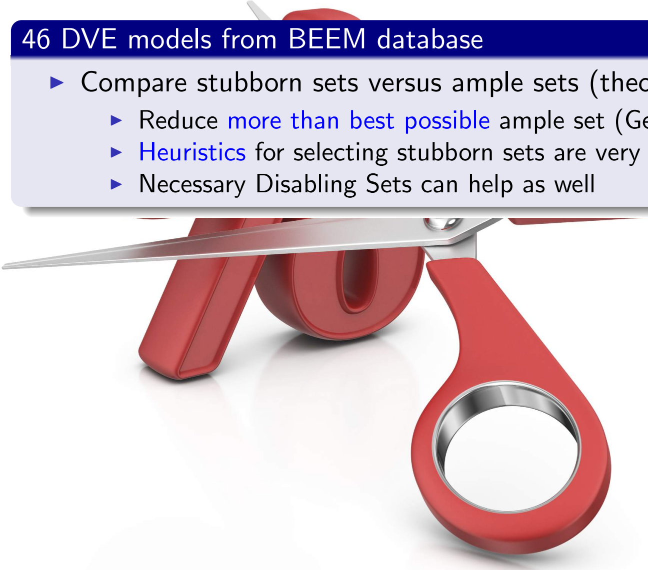
- Cycles: color proviso
  - Valmari, Evangelista
- Visibility: atoms as guards
  - Reuse en/dis-abling info
  - Dynamic (per state)

Experimental Results

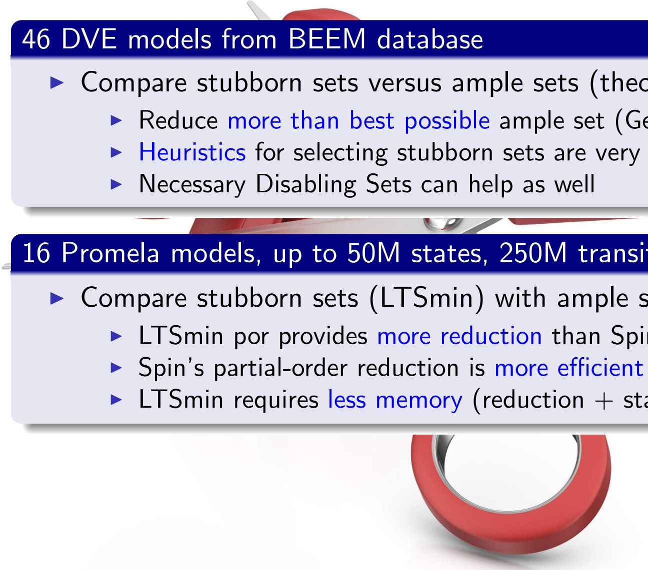
Experimental Results

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



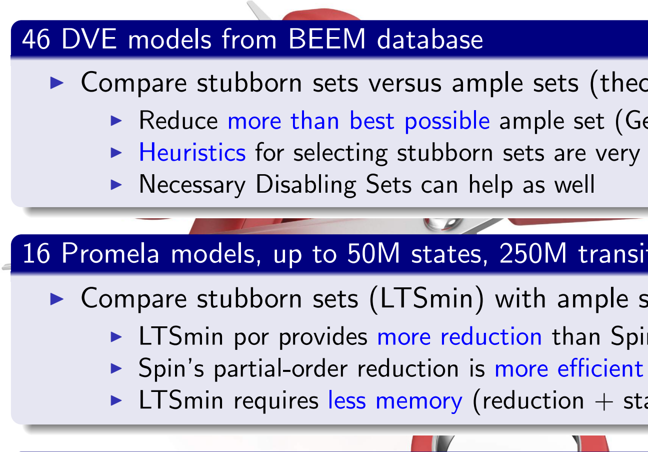
46 DVE models from BEEM database

- 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

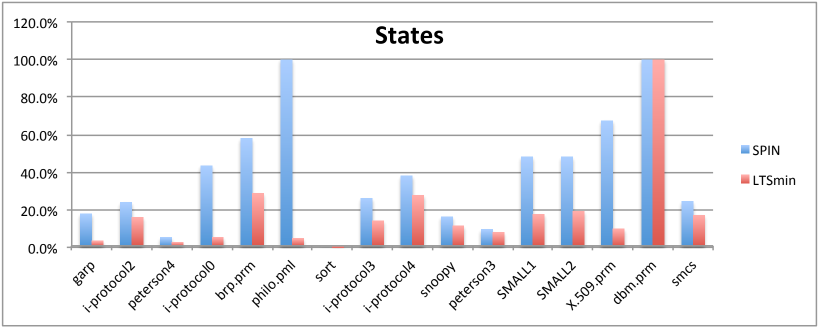
- Compare stubborn sets versus ample sets (theory):
  - Reduce more than best possible ample set (Geldenhuys)
  - Heuristics for selecting stubborn sets are very effective
  - Necessary Disabling Sets can help as well

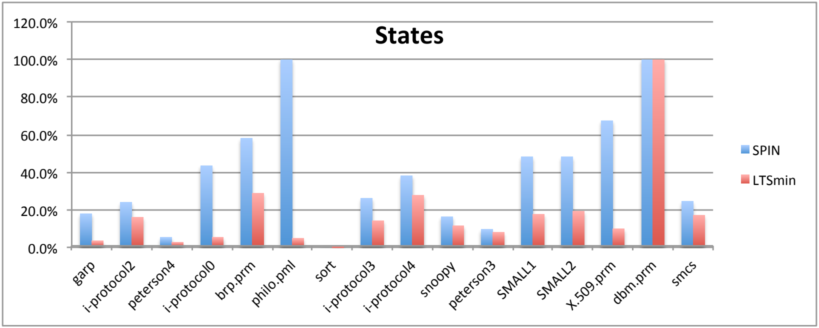
16 Promela models, up to 50M states, 250M transitions

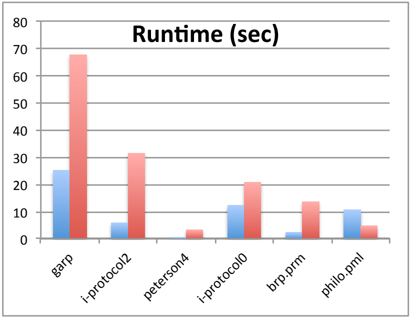
- Compare stubborn sets (LTSmin) with ample sets (SPIN)
  - LTSmin por provides more reduction than Spin por
  - Spin's partial-order reduction is more efficient in time
  - LTSmin requires less memory (reduction + state compression)

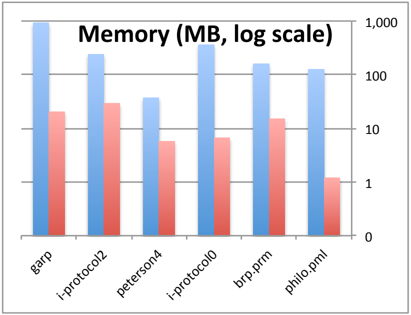
POR combined with LTL model checking

- Guard-based dynamic visibility proviso pays off
- Subtle cycle proviso's (Valmari, Evangelista) pay off

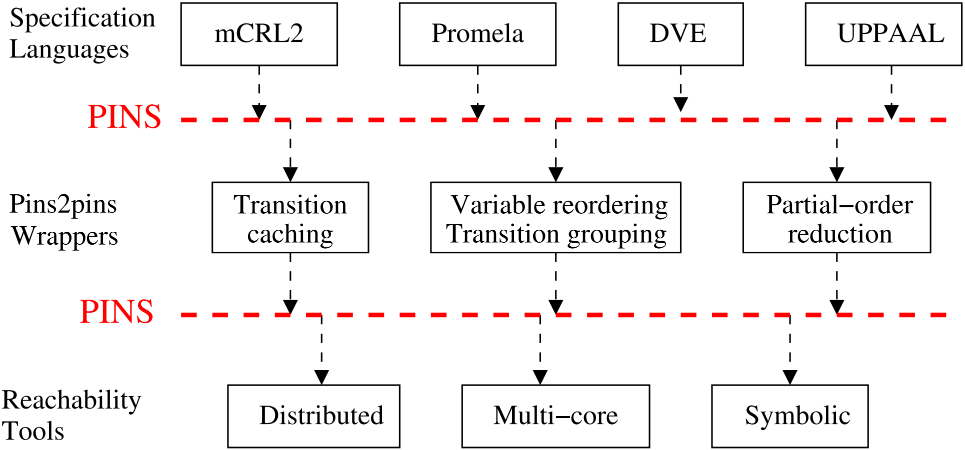

States
120.0%
100.0%
80.0%
60.0%
40.0%
20.0%
0.0%
garp
i-protocol2
peterson4
i-protocol0
brp.prm
philo.pml
sort
i-protocol3
i-protocol4
snoopy
peterson3
SMALL1
SMALL2
X.509.prm
dbm.prm
smcs
SPIN
LTSmin


States
120.0%
100.0%
80.0%
60.0%
40.0%
20.0%
0.0%
garp
i-protocol2
peterson4
i-protocol0
brp.prm
philo.pml
sort
i-protocol3
i-protocol4
snoopy
peterson3
SMALL1
SMALL2
X.509.prm
dbm.prm
smcs
SPIN
LTSmin


Runtime (sec)
80
70
60
50
40
30
20
10
0
garp
i-protocol2
peterson4
i-protocol0
brp.prm
philo.pml


Memory (MB, log scale)
1,000
100
10
1
0
garp
i-protocol2
peterson4
i-protocol0
brp.prm
philo.pml

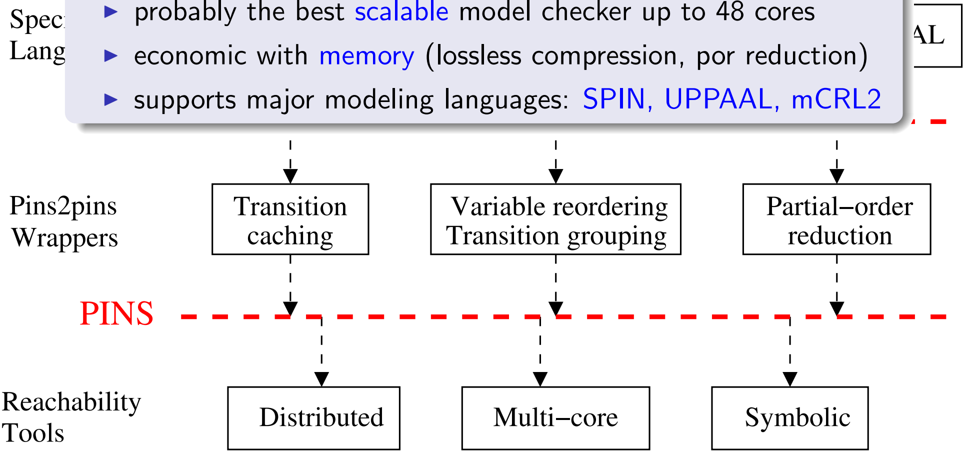
Why join the LTSmin project?



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



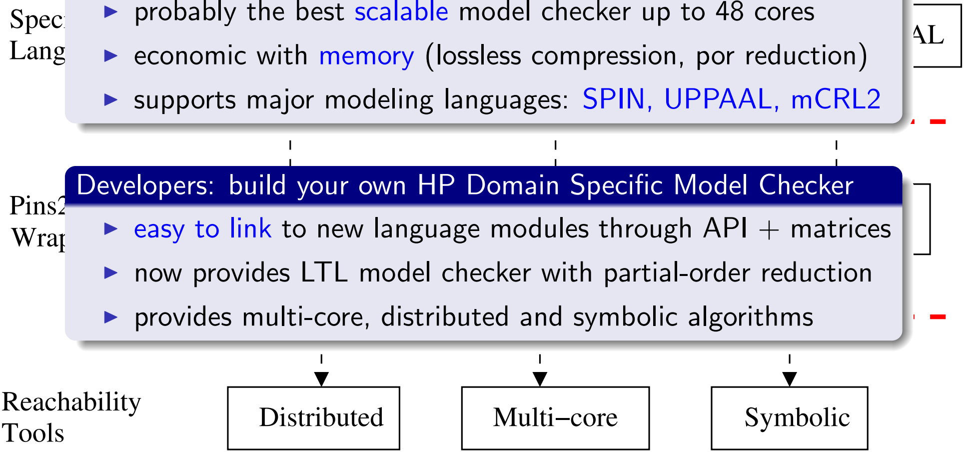
Why join the LTSmin project?

End users: profit without changing modeling language

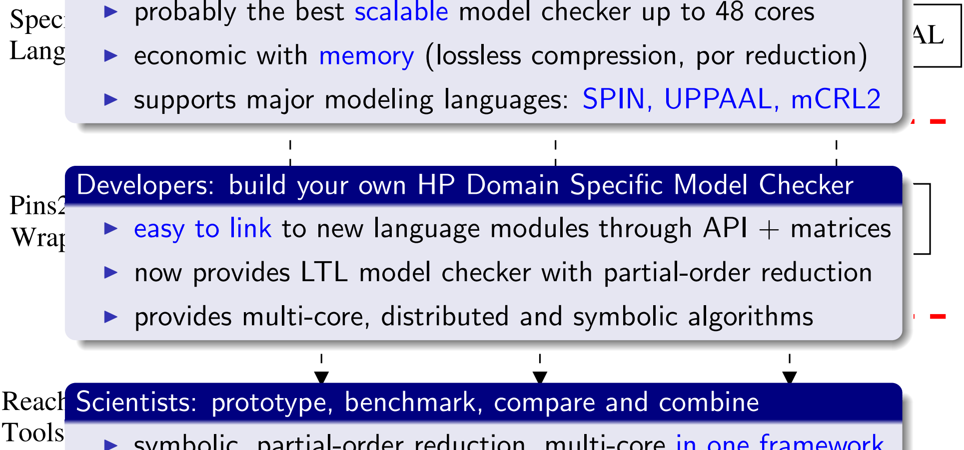
- probably the best scalable model checker up to 48 cores
- economic with memory (lossless compression, por reduction)
- supports major modeling languages: SPIN, UPPAAL, mCRL2

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



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