Petri Net (versus) State Spaces

Karsten Wolf
My experience with state spaces

-INA   Integrated Net Analyzer
-LoLA  A Low Level Analyzer
-The service-technology.org tool family

Case studies and applications:

- Finding hazards in a GALS wrapper
- Integration into Pathway Logic Assistent
- Soundness check for 700+ industrial business process models in (avg) 2 msec
- Verification of web service choreographies
- Verification of parameterized Boolean programs
- Solving AI planning challenges
- Integration into BP related tools like ProM, Oryx
- Integration into model checking platforms (MC Kit, PEP, CPN-AMI,…)
-….To be continued
Why state spaces?

Verification based on state space

Why Petri nets?
Why state spaces?

- Consider asynchronously communicating components rather than global state changes

- Consider causality of events rather than their ordering in time!
Petri net principles

- Consider asynchronously communicating components rather than global state changes
- Consider causality of events rather than their ordering in time!

Presence or absence of resources rather than reading / writing variables

Monotonicity of firing

Linearity of firing rule

Locality

Partially ordered event structures
Petri net specific verification

Monotonicity of firing

Coverability graphs
Siphons / traps

Linearity of firing rule

invariants

Locality

Net reduction

Partially ordered event structures

Branching prefixes
State space generation

1. Checking enabledness
2. Firing a transition
3. Backtracking
4. Managing the visited states
State space generation

1. Checking enabledness

   After firing, only check:
   previously enabled transitions which have lost tokens
   previously disabled transitions which have gained tokens
   ... managed through explicitly stored lists

   ... typical: reduction from linear to constant time

2. Firing a transition

3. Backtracking

4. Managing the visited states
State space generation

1. Checking enabledness

2. Firing a transition

   Marking changed via list of pre-, list of post-places
   - effort does not depend on size of net
   - Typically: constant effort

3. Backtracking

4. Managing the visited states
State space generation

1. Checking enabledness

2. Firing a transition

3. Backtracking

   In depth-first search: fire transition backwards

   In breadth-first search: implemented as incremental depth-first search

4. Managing the visited states
Consequence: „write-only“ storage of markings

Set of visited markings

Search stack

t1
t2
t3
...

current marking

-old/new

m

-fire

-fire backwards
4. Managing the visited states

only performed actions: search, insert

\[ a_1 p_1 + a_2 p_2 + a_3 p_3 = \text{const.} \]
\[ b_2 p_2 + b_4 p_4 + b_6 p_6 = \text{const.} \]
\[ c_3 b_3 + c_7 p_7 + c_8 p_8 = \text{const.} \]

30-60% less memory
preprocessing <1sec
run time gain: 30-60%
1. Linear Algebra
2. The Sweep-Line Method
3. Symmetries
4. Stubborn Sets
1. Linear algebra

• The invariant calculus
  – originally invented for replacing state spaces
  – in LoLA: used for optimizing state spaces

Already seen: place invariants

Transition invariant: firing vector of a potential cycle
Transition invariants

for termination sufficient: store one state per cycle of occurrence graph

implementation in LoLA:
transition invariants
  - set of transitions that occur in every cycle
  - store states where those transitions enabled

saves space, if applied in connection with stubborn sets, costs time
2. The sweep-line method

• Relies on progress measure

LoLA computes measure automatically:

\[ p_2 = p_1 + \Delta t_1 \]
\[ p_3 = p_2 + \Delta t_2 \]
\[ \ldots \]

transition invariant
3. The symmetry method

LoLA: A symmetry = a graph automorphism of the PT-Net

All graph automorphisms = a group (up to exponentially many members)
   - stored in LoLA: polynomial generating set

A marking class: all markings that can be transformed into each other by a symmetry
   - executed in LoLA: polynomial time approximation
... as derived from a program
4. Stubborn set method

- Dedicated method for each supported property

traditional LTL-preserving method:
- one enabled transition
- the basic stubbornness principal
- only *invisible* transitions
- at least once, on every cycle, all enabled transitions

LoLA:
- can avoid some of the criteria, depending on property
Conclusion

Why state spaces? Why Petri nets?

That's why

Further reading:
• Tools: www.service-technology.org
• Group / Papers: www.informatik.uni-rostock.de/tpp/