The SPIN Model Checker

Metodi di Verifica del Software

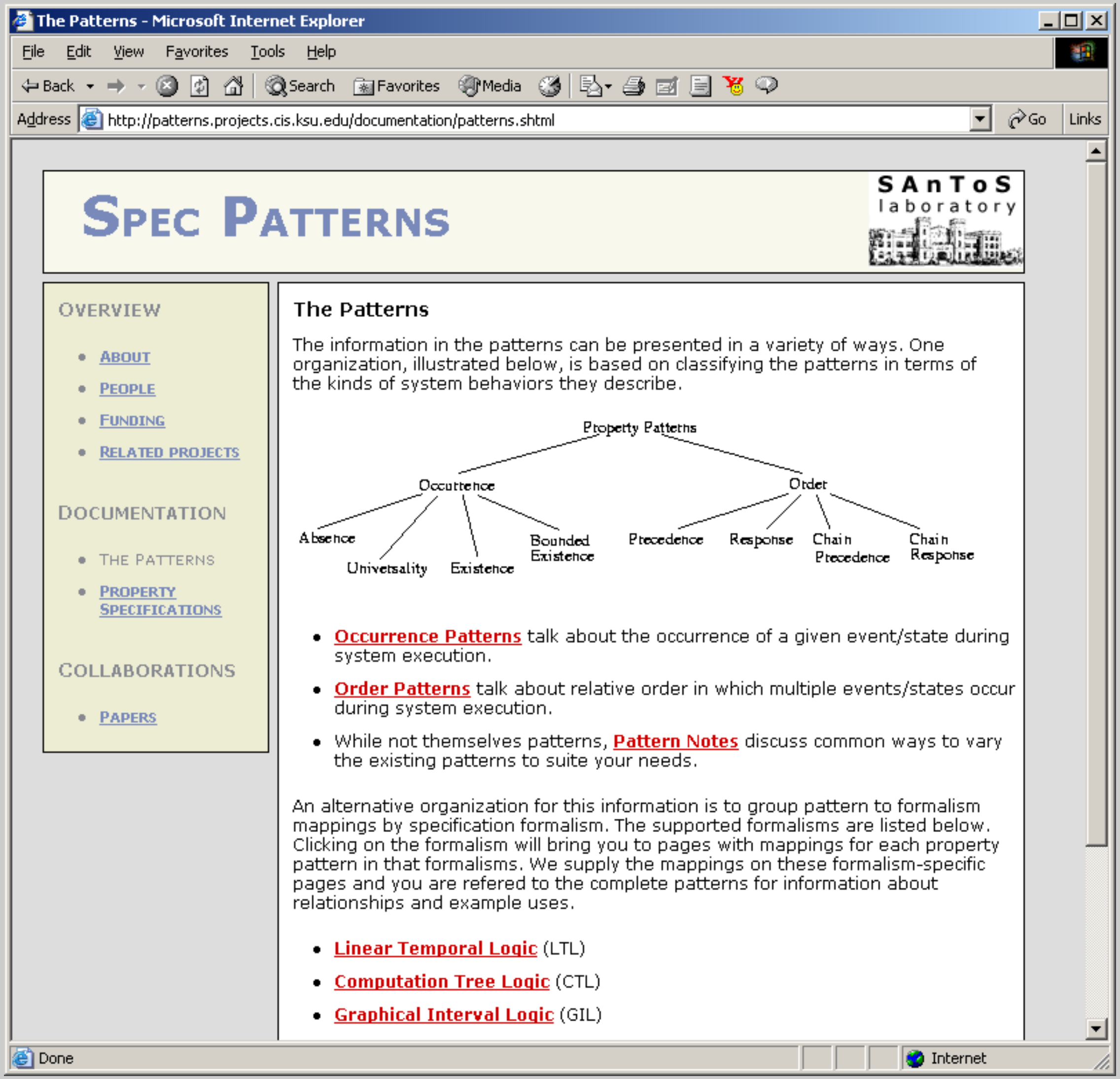
Andrea Corradini

Lezione 6

2013

Slides per gentile concessione di Gerard J. Holzmann

help with properties



SPEC PATTERNS

OVERVIEW

- ABOUT
- PEOPLE
- FUNDING
- RELATED PROJECTS

DOCUMENTATION

- THE PATTERNS
- PROPERTY SPECIFICATIONS

COLLABORATIONS

- PAPERS

The Patterns

The information in the patterns can be presented in a variety of ways. One organization, illustrated below, is based on classifying the patterns in terms of the kinds of system behaviors they describe.

Property Patterns
- Occurrence
  - Absence
  - Universality
  - Existence
  - Bounded Existence
- Order
  - Precedence
  - Response
  - Chain Precedence
  - Chain Response

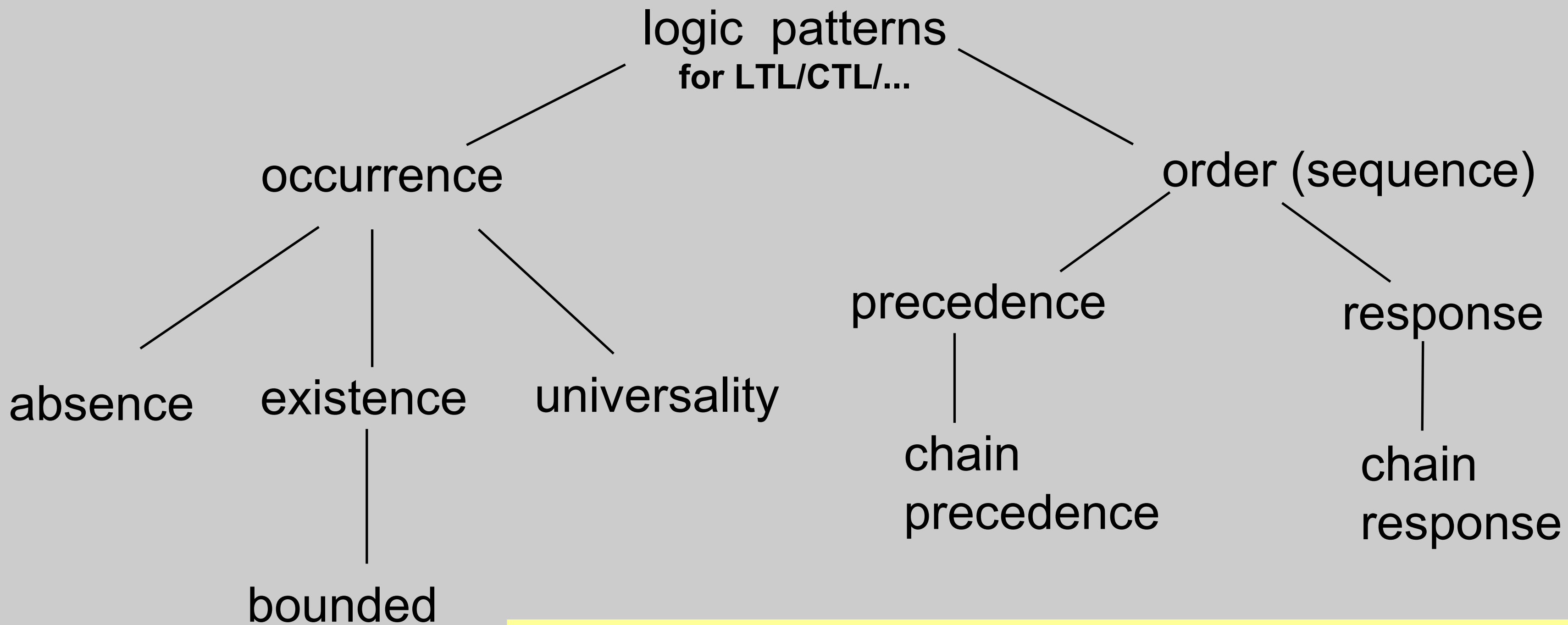
- **Occurrence Patterns** talk about the occurrence of a given event/state during system execution.
- **Order Patterns** talk about relative order in which multiple events/states occur during system execution.
- While not themselves patterns, **Pattern Notes** discuss common ways to vary the existing patterns to suite your needs.

An alternative organization for this information is to group pattern to formalism mappings by specification formalism. The supported formalisms are listed below. Clicking on the formalism will bring you to pages with mappings for each property pattern in that formalisms. We supply the mappings on these formalism-specific pages and you are refered to the complete patterns for information about relationships and example uses.

- **Linear Temporal Logic** (LTL)
- **Computation Tree Logic** (CTL)
- **Graphical Interval Logic** (GIL)

the temporal logic patterns database

http://patterns.projects.cis.ksu.edu/


logic patterns
for LTL/CTL/...

occurrence

absence

existence

bounded existence

universality

order (sequence)

precedence

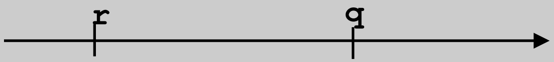
chain precedence

response

chain response

five variants are given for every pattern:

name	example for 'absence' and LTL	#states
globally !p	`[](!p)`	1
before r	`<>r -> (!p U r)`	4
after q	`[](q -> [](!p))`	2
between r and q	`[]((r && !q && <>q) -> (!p U q))`	4
after r until q	`[](r && !q -> ((!p U q) || []!p))`	4



expressiveness of LTL

compared to never claims

(cf. book p. 151)

- never-claims can define all ω-regular word-automata
- propositional linear temporal logic (without quantifiers) defines a *subset* of this language
  - anything expressable in LTL can be expressed as a never claim
  - but, never claims can also express properties that *cannot* be expressed in LTL
- adding a single existential quantifier over 1 propositional symbol to LTL suffices to extend its expressiveness to all ω-regular word-automata:

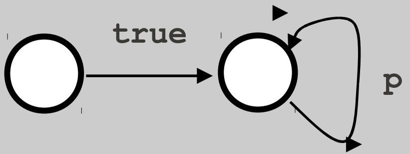
$$\exists p,\ [\,]\,(p \mathrel{-\!>} <>\ q)$$

- Kousha Etessami’s ‘temporal massage parlor’ TMP: http://www.bell-labs.com/projects/TMP

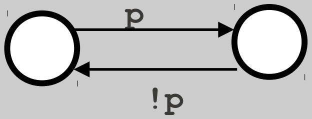
omega-regular properties

(~p. 150 book)

- something not expressible in pure LTL:
  - (p) *can* hold after an even number of execution steps, but *never* holds after an odd number of steps
  - [] X (p) certainly does not capture it:



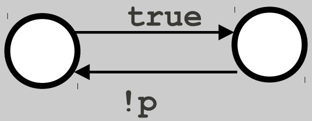
  - p && [](p -> X!p) && [](!p -> Xp) does not capture it either (because now p *must* always hold after all even steps):



(ltl2ba -f)

∃t, !t && [] (t -> X !t) && [](!t -> Xt) && [](p -> !t)

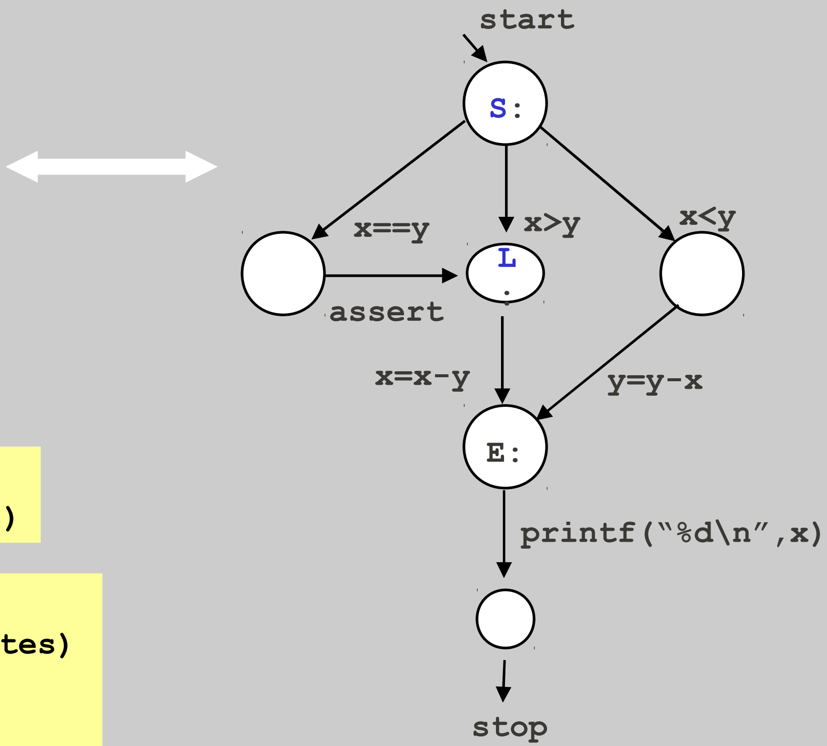
this formula expresses it correctly



On the semantics of Promela

proctypes and automata

```
active proctype not_euclid()
{
S: if
   :: x == y ->    assert(x != y); goto L
   :: x > y  -> L: x = x - y
   :: x < y  ->    y = y - x
   fi;
E: printf("%d\n", x)
}
```



a Spin model defines a system of:
   states and *state transformers* (transitions)

state is maintained in
   sets of process counters (*control* flow states)
   local and global variables and
   message channels

';', '->', if-fi, do-od, goto, etc. are only used to
   define the *transition structure*
   (not the state transformers themselves)
   the only *state transformers* are the *basic* statements:
      assignment, (expr), printf, assert, send, receive

operational model (see MVS page)

- to define the semantics of the modeling language, we can define an operational model in terms of *states* and *state transformers (transitions)*
  - we have to define what a “*global system state*” is
  - we have to define what a “*state transition”* is
    - i.e., how the ‘*next-state*’ relation is defined

- *global system states* are defined in terms of a small number of primitive objects:
  - we have to define: variables, messages, message channels, and processes

- *state transitions* are defined with the help of
  - basic statements that label transitions
    - the alphabet of the underlying automata
    - there are only 6 types of labels in the alphabet: assignment, condition, etc.
  - we have to define: transitions, transition selection, and transition execution

search algorithms in SPIN

- checking safety properties
  - basic depth-first search
  - variant1: stateless search [checks only the stack]
  - variant2: depth-limited search
  - breadth-first search
- checking liveness properties
  - non-progress cycles
  - acceptance cycles
  - Spin’s nested depth-first search algorithm
- fairness constraints
  - Choueka’s flag construction method
- optimization
  - partial order reduction, state compression, alternate state representation methods

basic depth-first search

```
Automaton A = { S, $s_0$, L, T, F }
Stack D = {}
Statespace V = {}

Start()
{
    Add_Statespace(V, A.$s_0$ )
    Push_Stack(D, A.$s_0$ )
    Search()
}

Search()
{
    s = Top_Stack(D)
    for each (s,l,s')$\in$ A.T
        if In_Statespace(V, s')== false
        {   Add_Statespace(V, s')
            Push_Stack(D, s')
            Search()
        }
    Pop_Stack(D)
}
```

Fig. 8.1 p. 168

`Push_Stack(D,s)`	adds s to *ordered* set D
`In_Stack(D,s)`	true iff s is in D
`Top_Stack(D,s)`	returns top element in D if any
`Pop_Stack(D)`	removes top element from D if any

`Add_Statespace(V,s)`	adds s to set V
`In_Statespace(V,s)`	true iff s is in V

the DFS is most easily written as a recursive procedure -- but the actual Spin implementation is iterative (originally to increase efficiency)

objective:
-store as little data about the graph as possible
- stores *states* in V, but not *transitions*
- Statespace V is there to prevent doing redundant work
- for correctness, V does not need to be complete
- in fact, V does not need to be there at all....

a stateless search

(memory efficient, but *excessively* time consuming...)

`no Statespace V`

```
Automaton A = { S, $s_0$, L, T, F }
Stack D = {}
/* Statespace V = {} */

Start()
{
    Push_Stack(D, A.$s_0$ )
    Search()
}

Search()
{
    s = Top_Stack(D)
    for each (s,l,s')∈ A.T
        if In_Stack(D, s')== false
        {   Push_Stack(D, s')
            Search()
        }
    Pop_Stack(D)
}
```

Fig. 8.5 p. 176

`replaced In_Statespace(V,s’) with In_Stack(D,s’)`

the algorithm is still guaranteed to terminate in a finite number of steps

Statespace V is used to prevent doing redundant work
- for correctness, it does not need to be complete
- in fact, it does not need to be there at all....

the nested depth-first search algorithm

```
Automaton A = { S, s0, L, T, F }
Stack D = {}
Statespace V = {}
State seed = nil
Boolean toggle = false

Start()
{   Add_Statespace(V, A.s0, toggle)
    Push_Stack(D, A.s0, toggle)
    Search()
}
```

```
Search()
{   (s,toggle) = Top_Stack(D)
     for each (s,l,s') ∈ A.T
     {    /* if seed is reachable from itself */
          if s' == seed ∨ On_Stack(D,s',false)
          {   PrintStack(D)
              PopStack(D)
              return
          }
          if In_Statespace(V, s', toggle) == false
          {    Add_Statespace(V, s', toggle)
               Push_Stack(D, s', toggle)
               Search()
    }     }
    if s ∈ A.F ∧ toggle == false
    {     seed = s   /* reachable accepting state */
          toggle = true
          Push_Stack(D, s, toggle)
          Search()  /* start 2nd search */
          Pop_Stack(D)
          seed = nil
          toggle = false
    }
    Pop_Stack(D)
}
```

enforcing fairness constraints

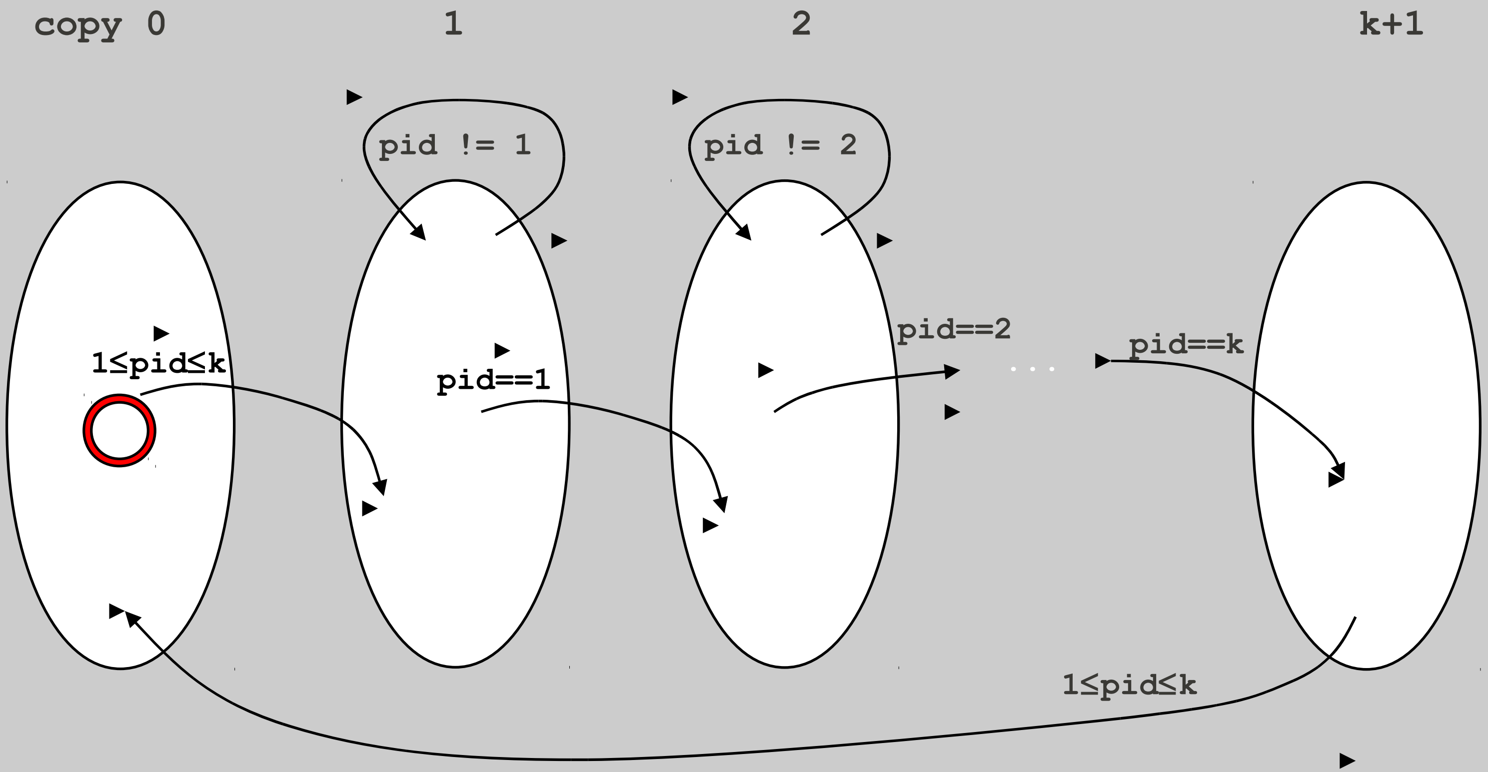
- fairness can be expressed in LTL, but this is not always simple / convenient
- we can also provide options in the model checker to enforce default types of process scheduling fairness
- there is a cost associated with the implementation as part of the nested depth-first search procedure:
  - weak fairness: linear increase of complexity (in # processes)
  - strong fairness: quadratic increase of complexity

the basic idea: unfolding

Choueka's flag construction method

- create (k+2) copies of the global reachability graph, with k the number of active processes
  - we number them from 0..(k+1)
- preserve accept-state labels only in the 1st copy
  - the copy numbered 0
- change the transition relation to connect all k+2 copies:
  - in copy 0, change the destination state for outgoing transitions of all *accepting* states so that they point to the corresponding state in copy 1
  - in copy k+1, change the destination state for outgoing transitions of *all* states so that they point to the corresponding state in copy 0
  - in copy *i*, 1≤*i*≤k, change the destination state for all transitions contributed *by process i* to the corresponding state in copy *i*+1
  - add a *nil*-transition from any state in copy *i* where process *i* is blocked (has no enabled transitions) to the same *state* in copy *i*+1
- an accepting ω-run in the unfolded graph now necessarily contains transitions from *all* active processes and therefore satisfies the weak fairness requirement

(k+2)-times unfolded graph



all runs of the original system are preserved, but unfolded.
no accept cycles can exist *within* copy 0
all accept cycles must traverse all copies to return to copy 0
and are therefore necessarily weakly fair

fair reminders

- Spin’s built-in notion of fairness applies only to
  - weak fairness, not strong fairness
  - process scheduling
  - not to non-deterministic choices within a process

- other types of fairness can be expressed in LTL with properties of the type []<>p

relative complexity

- parameters:
  - k processes – typical values: 2..10
  - M reachable states in model – typical values: 10^9 .. 10^{11}
  - B states in property automaton – typical values 1..4
  - S size of one state in bits

`problem size`
`P = (M*B*S)`

	Memory	Run-Time
safety properties	P	P
liveness properties	P	P*2
liveness+weak fairness	P	P*2*(k+2)

```
use abstraction and p.o. reduction to keep the model size M small
use abstraction and compression to keep state size S small
use simple properties, exploit separability, to keep B small
use safety properties when possible
    liveness only when needed
    fairness constraints only when unavoidable
```

search optimization

- the complexity is determined by M*B*S: reducing any of these 3 numbers reduces verification complexity
  - M: numbers of reachable states in the global state space
    - the size of the asynchronous product automaton
  - B: the number of states in the property automaton

- M dominates (typically 10^6 states and up), B is almost always very small (1..6 states)
  - M *can* increase exponentially with the number of asynchronous processes and message channels in the model
    - in many cases this can be avoided by revising the model slightly
    - reducing the nr of processes and/or data objects, splitting data streams
  - B *can* increase exponentially with the number of sub-formulae (or roughly: the number of operators) in an LTL formula
    - in practice this is insignificant compared to the other factors that contribute to complexity
    - see Appendix B re comparisons between CTL/LTL

non-algorithmic techniques to reduce complexity

- to reduce M*B*S
  - B: reducing the size of the property automaton
    - use small separable properties, instead of one large combined one
  - M: reducing the size of the global state space
    - reducing the number of processes, message channels, data objects
    - reducing the length of channels (number of slots)
    - use a unique channel for each sender-receiver combination
    - avoid data types with larger than necessary range
    - using abstraction, separation of concerns, generalization, etc.
  - S: reducing the size of individual states (the state-vector)
    - using abstraction, lossless or lossy compression, or alternate state representation methods

algorithmic techniques to reduce complexity

- to reduce M: partial order reduction (default in Spin)
  - avoids computing equivalent paths and states
- to reduce S:
  - lossless compression
    - masking unused parts in state-vector (default in Spin)
    - collapse compression (-DCOLLAPSE), increases time, reduces memory
  - lossy compression
    - hash-compact (-DHC), no increase in time, reduction in memory use, modest risk of incompleteness
    - bitstate hashing (-DBITSTATE), reduction in time, large reduction in memory use, risk of incompleteness (statistical estimates of coverage)
  - indirect methods
    - using a recognizer (a minimized automaton) instead of a hashed lookup table to store states (-DMA), major increase in time, major reduction in memory