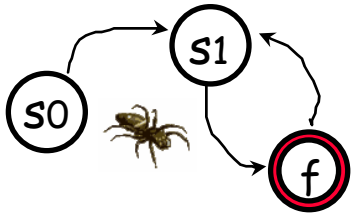


Automata-based Model Checking

Gerard J. Holzmann
Computing Principles Research
Bell Labs, USA

outline

- introduction
  - what is model checking all about
  - what are the central issues
- the automata theoretic method
  - finite automata and omega automata
  - relation with propositional linear temporal logic
- spin's core algorithms for ltl model checking
  - depth first search, nested depth first search
  - state storage, memory/time trade-offs
- practical considerations
  - complexity and abstraction
  - some applications

1. introduction

designing systems

- "a design without requirements cannot be right or wrong, it can only be surprising"
- the designer's first task is to find out what the requirements are, and then to build a system that meets those requirements

Design = Specification + Requirements

- design verification = showing that the specification logically implies the requirements

principles of design

- principles of design used in most disciplines:
  - abstraction
    - "modeling" (constructing mathematical models, and/or executable models/design prototypes)
  - analysis
    - "model checking" (manual or automated)
- principles used in software engineering:
  - trial and error
    - running it until it appears to work...
  - duplication
    - adjusting trusted earlier designs by peers
- model checking allows the first set of principles to be applied in the design of distributed systems software

model checking

- to prove that a system implies its specification:

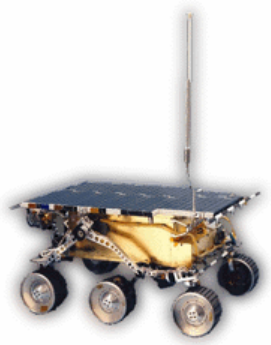
1. construct a model of the system (an *abstraction*)
2. formalize its essential properties (e.g., in PLTL)
3. run the model checking algorithm
4. if answer is inconclusive, revise 1 or 2 and repeat 3 and 4

- the possible answers provided by a model checker
  - ***ok*** -- if the model definitely satisfies the property
  - ***not ok*** -- if not, illustrated by a counter-example
  - ***inconclusive*** -- if resource-limits (time/space) prohibit the algorithm from running to completion

state implosion

- model checkers work by the grace of the fact that logical properties of bounded models are decidable
  - more about the complexity bounds later
- time and space are also bounded
  - time: a hard upper-bound is somewhere around 76x364x24x60x60 CPU seconds (~ 10^9)
  - space: today, somewhere around 10 Gbyte (~ 10^{10})
- the bounds of the model can easily exceed the bounds imposed by physical reality….
- in practice, the art of model checking is the art of building *executable abstractions*

modeling = abstracting

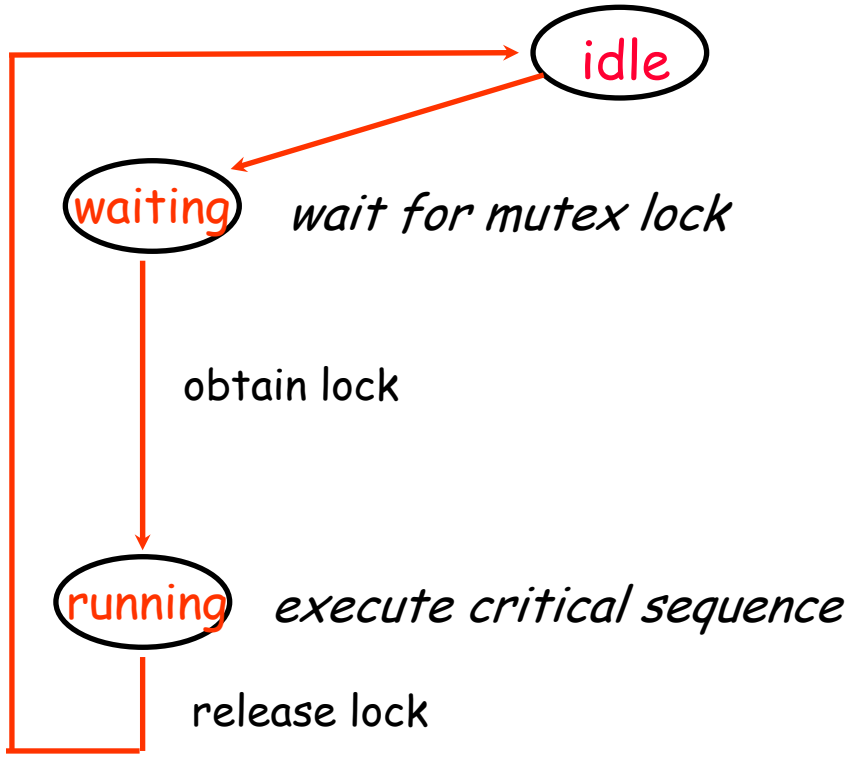


a small example:

The requirements for NASA's Mars Pathfinder control software included:

- mutual exclusion rules
  - a process cannot access the databus unless it owns a mutual exclusion lock
- scheduling priority rules
  - a lower priority process cannot execute when a higher priority process is ready to execute
  - saving data to memory, for instance, has higher priority than processing data

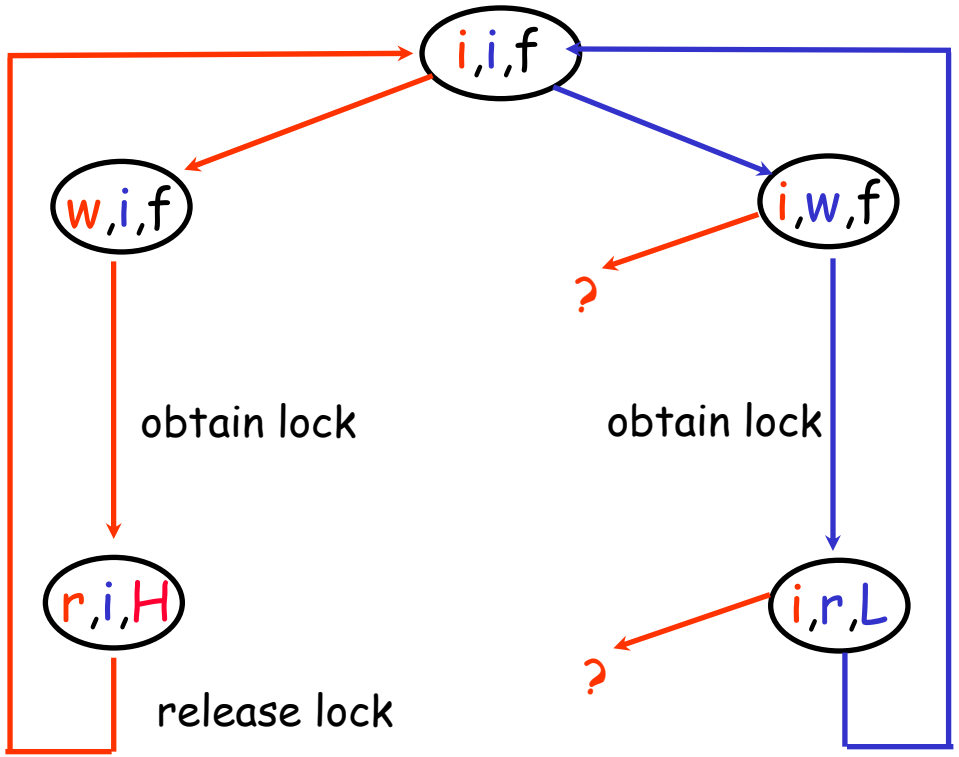
basic sequence for data-bus access



enforcing the priority rule

i = idle
w = waiting
r = running

f = mutex lock free
H = high owns lock
L = low owns lock



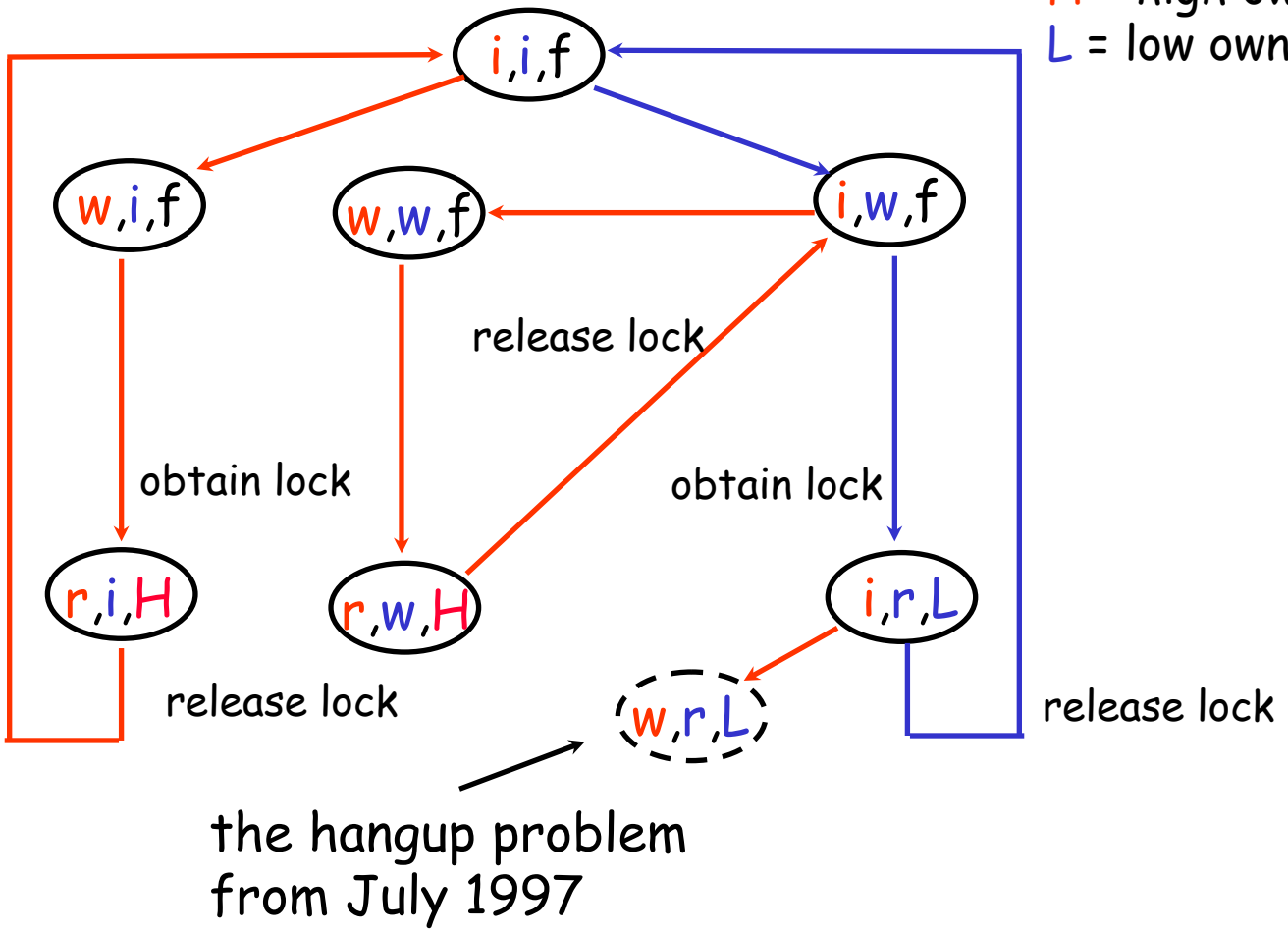
High priority process
(dumping measurements)
restricted by lock

Low priority process
(processing data)
cannot run unless: i,-,-

model checking run (e.g., []<> (i,i,f))

i = idle
w = waiting
r = running

f = mutex lock free
H = high owns lock
L = low owns lock



synopsis

the model checking challenge is to find a model that is as simple as possible, but no simpler...

"Seek simplicity -- and distrust it"

Alfred North Whitehead (1861-1947)

"Entities should not be multiplied unnecessarily"

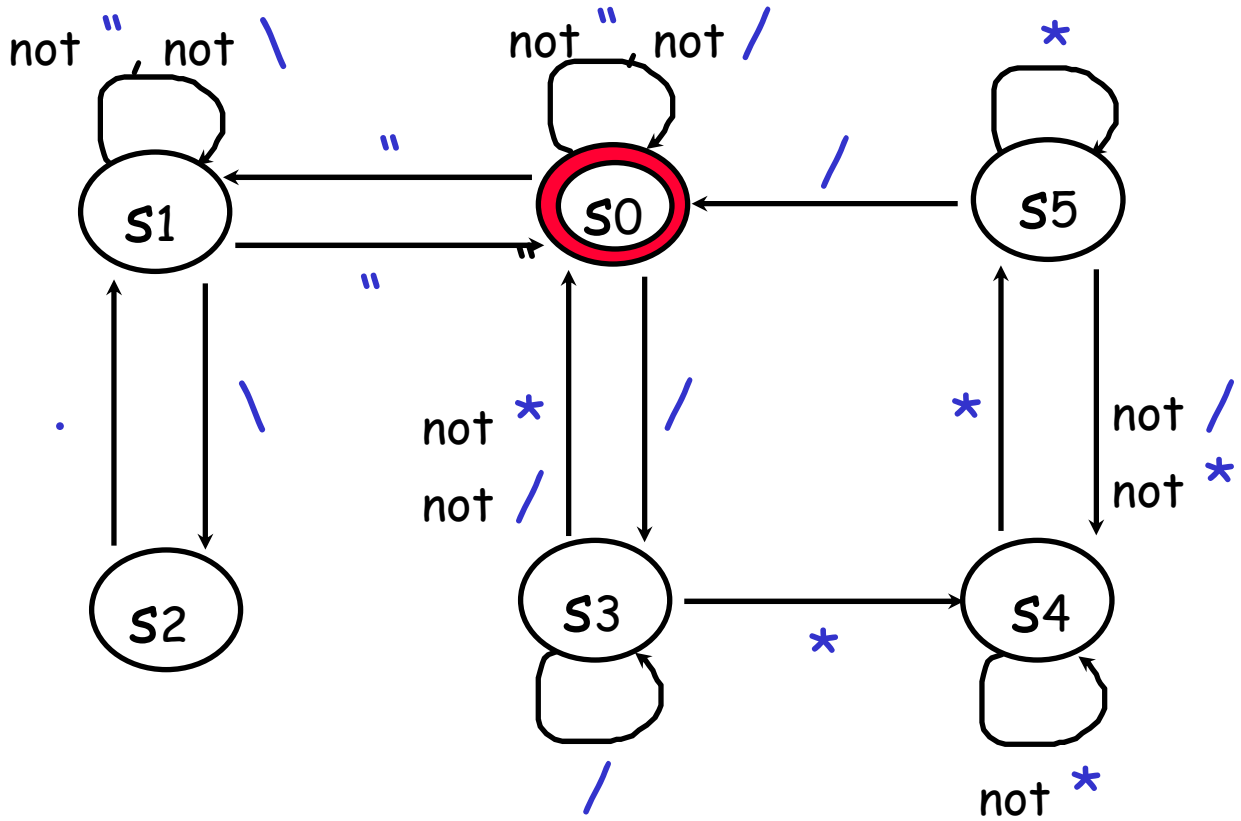
William of Occam (c. 1285-1349)

2. the automata theoretic method

finite automata (labeled transition systems)

- a classic FSA is a tuple {S, s0, L, T, F} with
  - S -- a finite set of states
  - s0 -- a distinguished initial state
  - L -- a finite set of labels / symbols
  - T -- a set of transitions from SxLxS
  - F -- a set of final states from S
- a run of an FSA is an ordered set of transitions from T ..., {si,li,si+1}, {si+1,li+1,si+2}, ... such that
  - for all i >= 0: {si,li,si+1} is in T
- an accepting run of an FSA is a finite run in which the final state is in F

a sample fsa



(FSA for checking C-style comments /* ... */)

the notion of acceptance

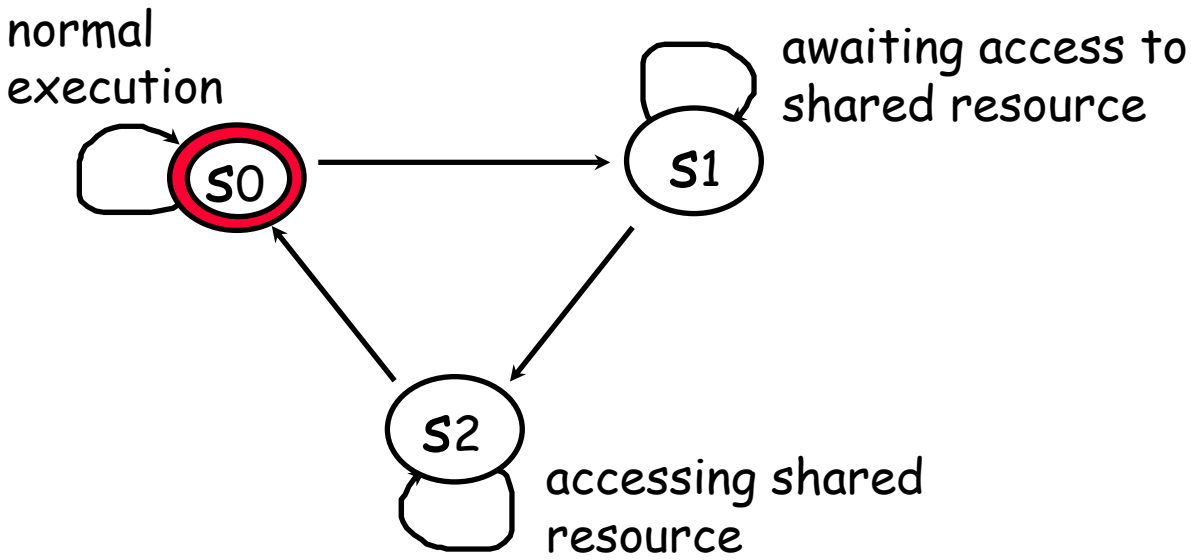
- an ω-run of an FSA is a run that is infinitely long
  - an acyclic FSA does not permit ω-runs
- an ω-run is accepting if at least one state in F appears infinitely often within the run
  - this is called Buchi Acceptance
  - automata with this acceptance rule are Buchi Automata
- some decidable properties of Buchi automata :
  - intersection (given two Buchi automata, find one automaton that accepts only runs accepted by both)
  - union (given two Buchi automata, find one automaton that accepts all runs accepted by either)
  - non-emptiness (does a Buchi automaton accept any ω-runs at all?)

stuttering

- Problem: if an FSA has both finite and infinite runs -- can we apply a notion of ω-acceptance to both?
- Stutter extension rule:
  - the final state of a finite run is considered to be repeated ad infinitem on a dummy self-loop

    ω-acceptance is applied as before
- Stutter invariance:
  - a property is stutter invariant if it is insensitive to the introduction or removal of stuttering steps at any point in the execution

a sample buchi automaton



(only runs that do not get stuck in states s1 or s2 are to be accepted)

automata theoretic verification

vardi/wolper [vw86], given:

- a system modeled by ω-automaton R
- a correctness requirement given as PLTL formula f

then

- convert f into an ω-automaton P that satisfies !f (the negation of f)
- compute the language intersection V = P ⇕ R

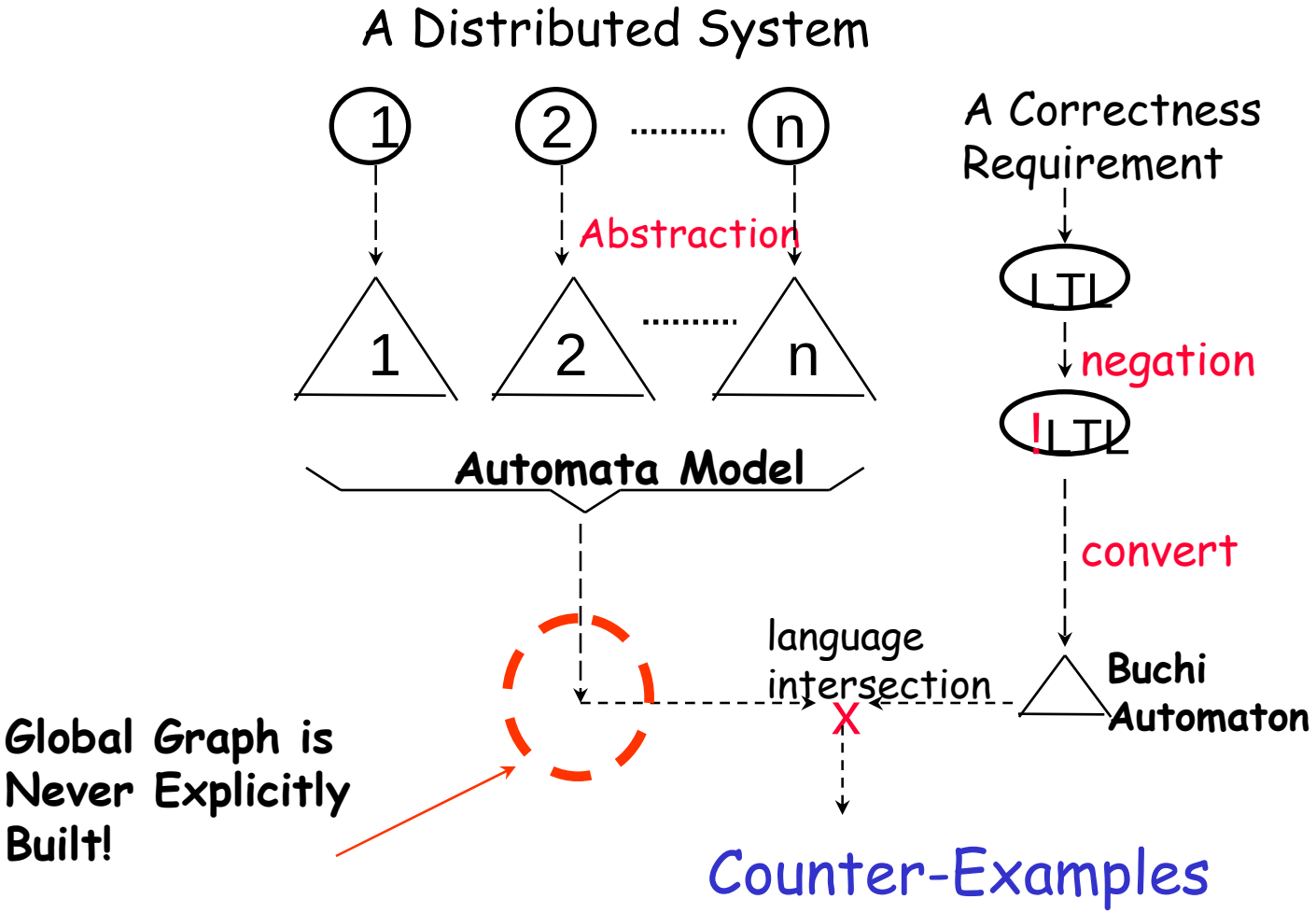
result

- if V is empty, f is satisfied
- if not, V contains the violations of f

application challenges

- conversion from a program-like notation into formal automata
- computation of a global automaton from local automata, given
  - asynchronous process components
  - dynamic creation and deletion of processes
  - message passing and rendezvous semantics
- it turned out to be a natural fit for Spin
  - a general-purpose verification system based on automata
  - using an on-the-fly verification paradigm (1980)

Spin's on-the-fly verification procedure



computational cost

- complexity bounds:
  - LTL model checking is PSPACE-complete [Sistla&Clark85]
- verifying safety properties (states):
  - linear in the number of reachable states R
- verifying liveness properties (cycles):
  - at most 2xR time, and R space (memory)
- verifying ω–properties (including ltl):
  - with property automaton of N states:
  - 2xRxN time, and RxN space
  - for LTL, N may be exponentially larger than the number of operators in the formula: $1 <= N <= 2_{|f|}$
  - in practice N is typically in the range 1..10

practice: two ways to build a verifier

- **Given**

  L = language of the system

  P = language of the requirement

- **First** Method

  prove: P includes L

  assuming P *holds,* minimal cost to prove this by language inclusion is **P + L**

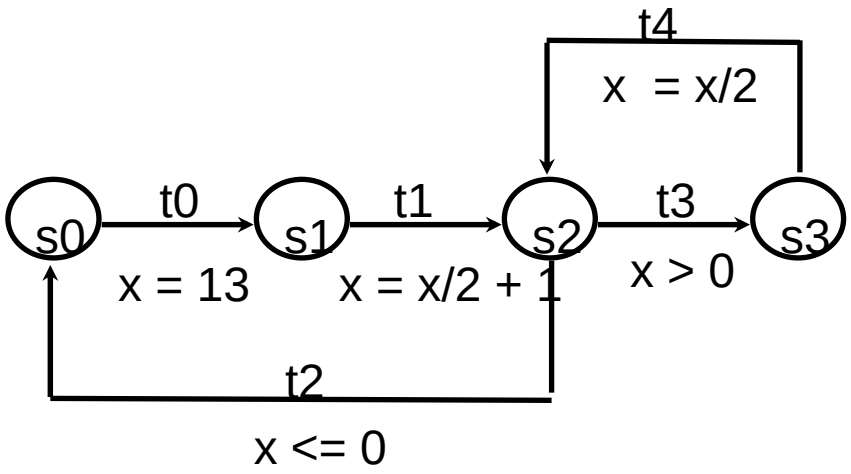
- **Second** Method

  construct !P

  prove: P excludes L

  assuming P *holds,* minimal cost to prove it by language intersection *is* **zero**

correctness: reasoning about behavior



3 views of an execution:

a sequence of states: s0 s1 s2 s3 s2 s3 s2 s3 s2

a sequence of events: t0 t1 t3 t4 t3 t4 t3 t4 t2

a sequence of properties: F T T T T T T F F

(x%2 > 0)

from logic to automata

- The truth of an LTL formula is defined over execution sequences
  - the 3rd view of an execution applies: a sequence of propositional values
- For any LTL formula f there exists an automaton that accepts precisely those executions for which f is true
- If the automaton is insensitive to stuttering, it can be very small
- The acceptance conditions of the automaton are defined over infinite sequences (Buchi acceptance)
- Example: the LTL formula <>[] P corresponds to the Buchi automaton:

temporal operators in ltl

the until operator:

- (p U q) -- p holds at least until q holds
- defined over ω-runs -- there are two variants:

σ[i] |= (p U q)

weak until

<=>

σ[i] |= q or

σ[i] |= p and σ[i+1] |= (p U q)

σ[i] |= (p U q)

strong until ← spin

<=>

(p U q) and ∃j, j >= i, σ[i] |= q

always, eventually:

σ[i] |= ([] p) <=> (p U false)

σ[i] |= (<> p) <=> (true U p)

spin's LTL grammar

- ltl formula f ::=
  - true, false
  - propositional symbols p, q, r, …
  - (f)
  - unary_operator f, f binary_operator f
- unary_operators:

  `[]` --- always, henceforth (*box*)

  `<>` --- eventually (*diamond*)

  `!` --- logical negation (*not*)
- binary_operators:

  `U` --- strong until

  `&&` --- logical and

  `||` --- logical or

  `->` --- implication

  `<->` --- equivalence

ltl formulae

- two useful equivalences:

  ![] p <=> <> !p

  !<> p <=> [] !p

- a deliberate omission (to secure stutter invariance):
  - X --- the next operator
- often used properties:
  - [] p --- invariance
  - <> p --- guarantee
  - p -> (<> q) --- response
  - p -> (q U r) --- precedence
  - []<> p --- recurrence (progress)
  - <>[] p --- stability (non-progress)
  - <> p -> <> q --- correlation

logic -> automata

manual conversion:

! [] (p –> <> q)

! [] (!p \/ <> q)	the definition of ->
<> ! (!p \/ <> q)	equivalence
<> (p /\ !<> q)	De Morgan's law
<> (p /\ [] !q)	equivalence

the corresponding Buchi Automaton:

(don't worry, there's an easier way…)

deriving automata from ltl formulae

- define the *Closure* of f:
  - Cl(f) = the set of all sub-formulae of f and their negations
- example:
  - Cl (<> [] P) =

    <> [] P

    <> P

    P

    !<> [] P

    !<> P

    !P

step 1: construct a generalized buchi automaton

- $A = \{S, s0, L, T, F\}$
  - States: $S = 2^{Cl(f)}$
  - Initial state: s0, with formula $f \in s0$
  - Labels (alphabet): $L = 2^{Prop}$
  - Transition relation: T from S x L x S
  - Acceptance: n sets F_i, subsets of S
- Define the transition relation as:
  - $(s, p, s') \in T$ iff
  - (p) satisfies s
  - $(q \; U \; r) \in s$ implies
    - $r \in s$ *or* $q \in s$ and $(q \; U \; r) \in s'$

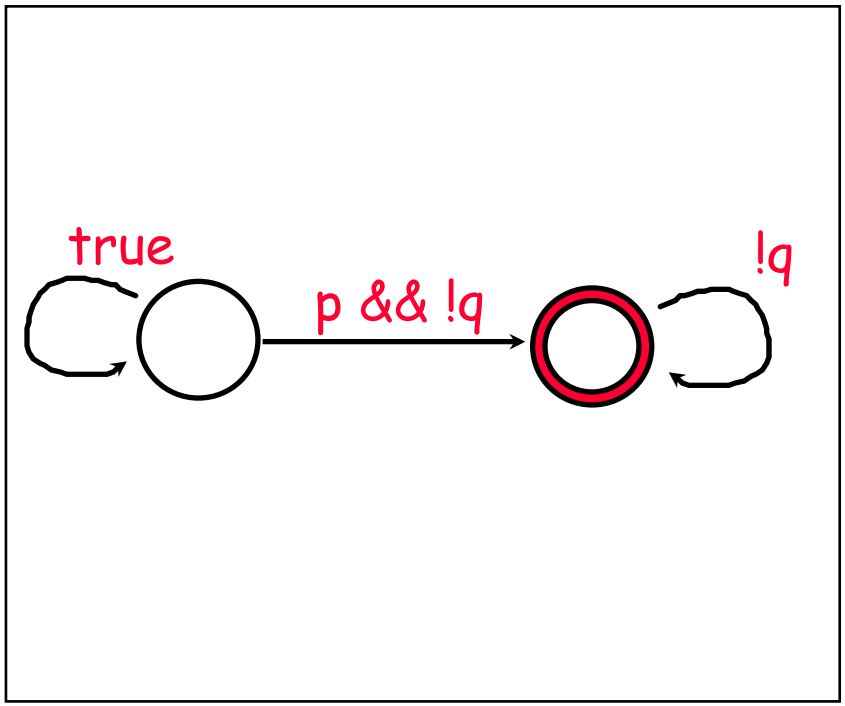
    (i.e., (qUr) is satisfied at s or it persists)
- For each subformula (q U r):
  - $F_i = \{s : (q \; U \; r) \notin s \text{ or } r \in s\}$

step 2: map the result to a standard Buchi automaton

- generalized Buchi acceptance defines:
  - *k* sets F: F*1*, …, F*k* and
  - For *each* set *i*, require infinitely many visits in F*i*.
- Choueka's flag construction can be used to translate *generalized* into *standard* Buchi acceptance [Choueka,1974]
  - Add counter n, with initial value 0
  - Upon reaching any state in F*i*, increment n = (n+1) % k (with k the number of acceptance sets F*i*)
  - Select any one of the sets F*i* as the new single acceptance set F

spin's syntax for ω-automata

ω-automaton:



corrresponding never-claim:

```
never {
        do
        :: true
        :: p && !q -> break
        od;
accept:
        do
        :: !q
        od
}
```

- Other types of acceptance can also be expressed in never claims

Rabin acceptance [Rabin72]

- Define n pairs of set of states {Li,Ui}
- for at least one i, detect any infinite execution sequence with
  - infinitely many visits in Ui
  - but only finitely many visits in Li
- Propositional symbols ui, li formalize the corresponding sets

```
never {
            do
            :: true                          /* allows li  */
            :: u1 && !l1 -> goto set1        /* choice   */
            :: u2 && !l2 -> goto set2
            :: ...
            od;
set1:       do
            :: !l1
            :: u1 && !l1 -> accept1: skip
            od;
set2:       ...
}
```

3. Spin: core algorithms for ltl model checking

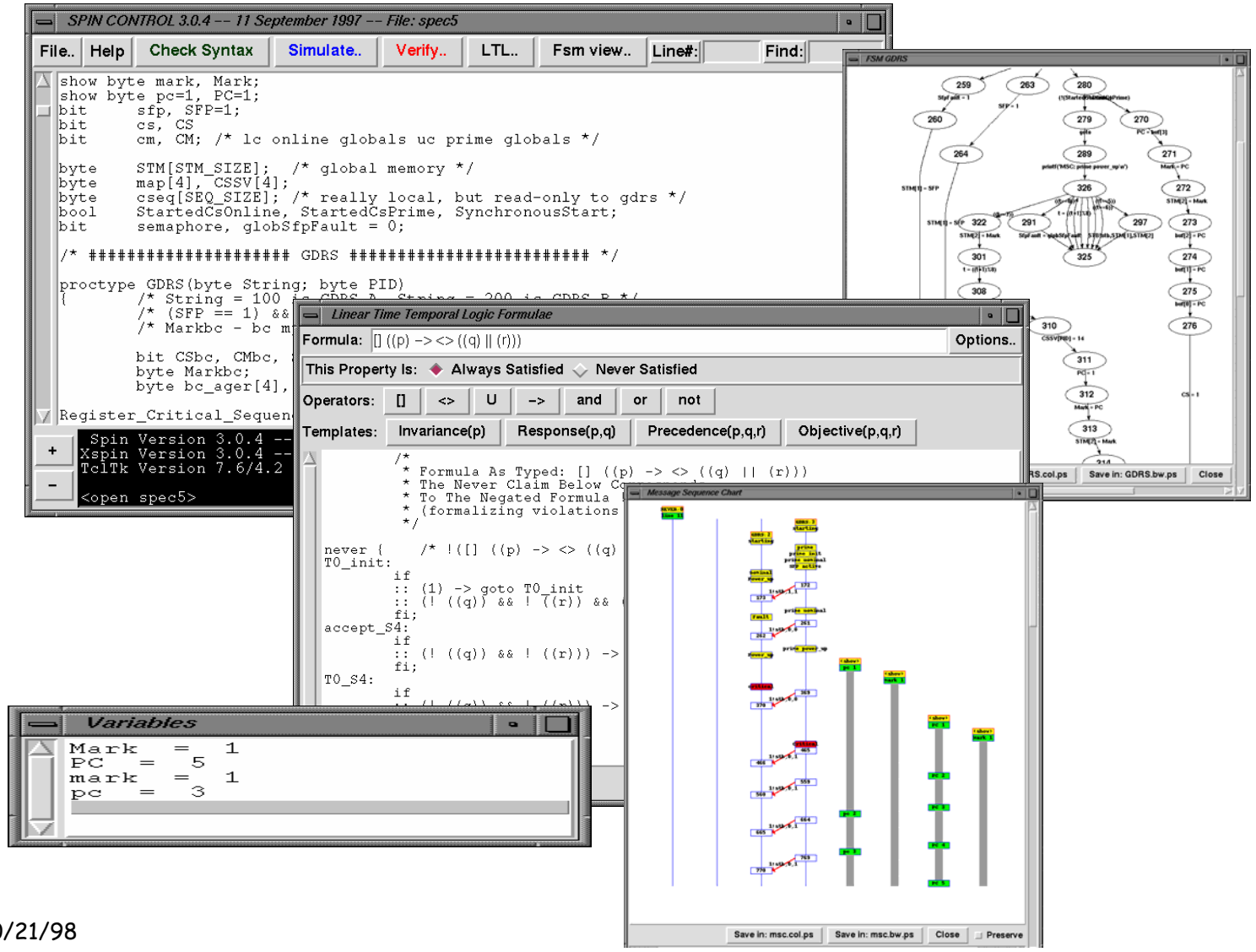
choices, choices...

- **the focus of the exisiting model checkers differs significantly**
  - they are optimized for different domains of application and different logics
  - making it hard to do head-to-head comparisons
  - each model checker can soundly beat the others within its own domain of application...
- **model checkers are intended to be applied to bounded models, not directly to implementations**
  - a model is a design abstraction
  - properties of bounded models are decidable
  - models and ltl properties can be converted mechanically into finite automata and can be analyzed in that domain

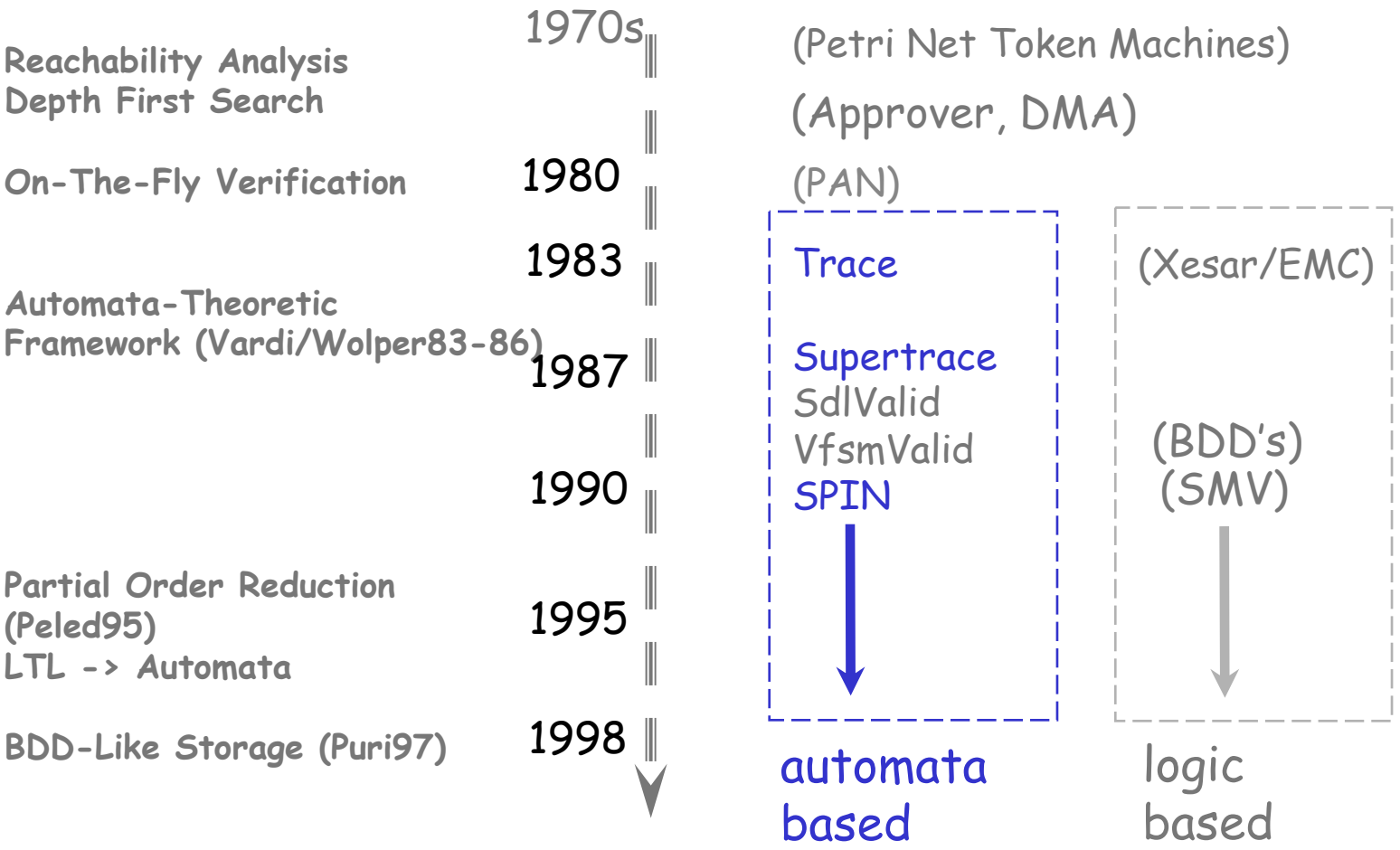
spin's focus

- **Spin models describe distributed, not sequential, systems**
  - they focus on **coordination** not **computation**
- **Spin targets asynchronous, not synchronous, systems:**
  - Spin targets software, not hardware, verification problems
  - it uses primitives for message-channels, rv-ports, etc.
  - not gates, clocks, signal wires, flip-flops, etc.
- **Spin is optimized for the verification of omega regular properties,** with LTL as its most significant subset.
  - It cannot handle branching-time properties.

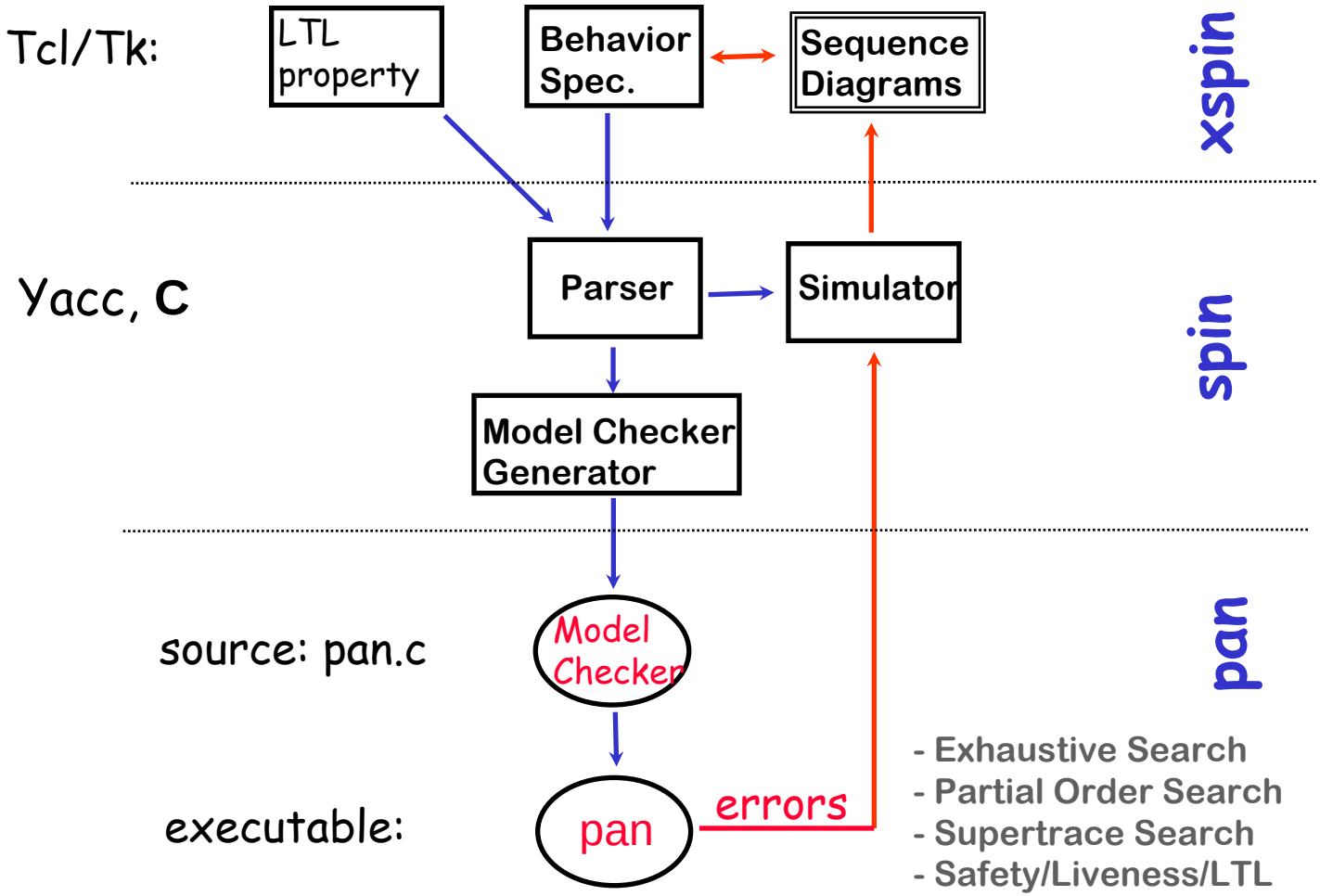
spin's graphical user interface



spin's origin (and automata-based verification)



spin's structure

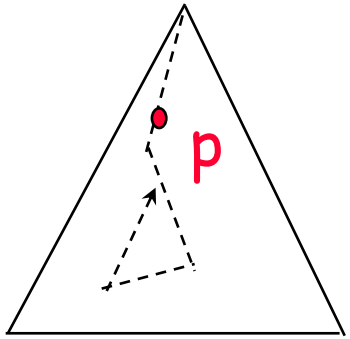


on-the-fly cycle detection

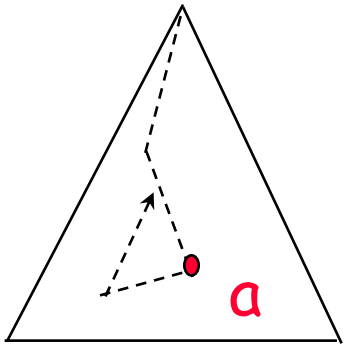
- violations of ω–properties correspond to executions that contain infinitely many accepting states
- in a finite graph, these correspond to reachable strongly connected components (scc's) with one or more accepting states
- Tarjan's classic depth-first-search algorithm could be used
  - cost: linear in the size of the graph
- but, it suffices to prove or disprove that:
  - no reachable accepting state is reachable from itself…
  - has the same complexity, but a lower memory overhead

non-progress and acceptance

- non-progress:
  - an ω–run, containing only finitely many progress states
  - in ltl, using buchi acceptance: <>[] np_
- buchi acceptance:
  - an ω–run, containing at least one accepting state infinitely many times



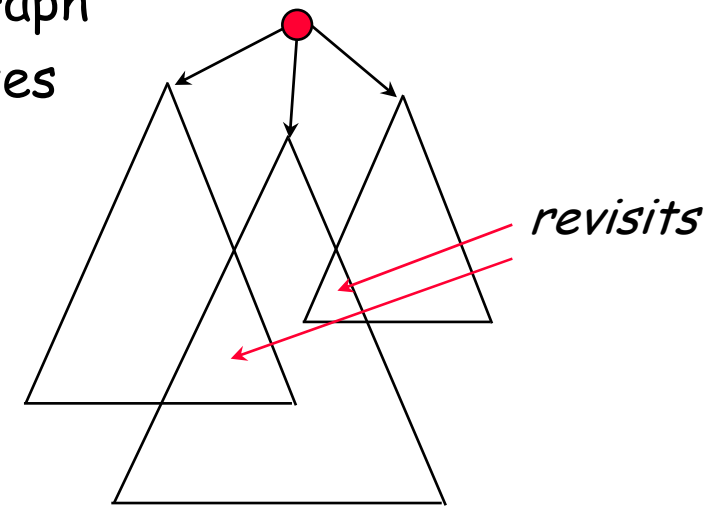
non-progress



acceptance

depth-first search

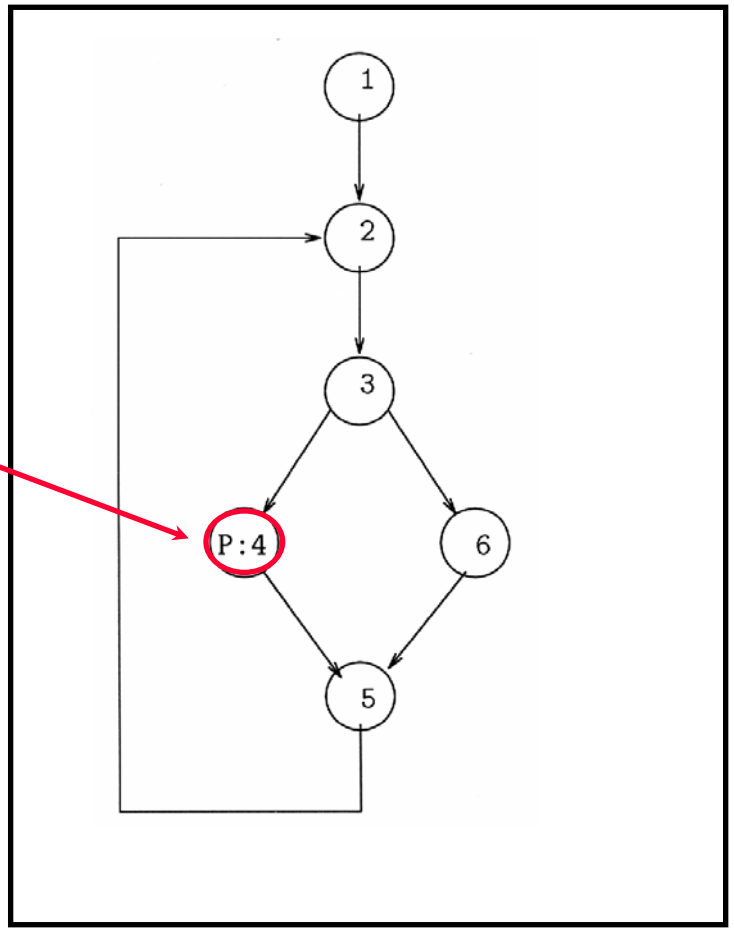
- recursive expansion of global graph
- asynchronous interleaving of concurrent actions
- storing as little data as possible about previously visited states
  - need only detailed info for counter-examples, not for global graph
  - no need to store edges of graph
  - one-way compression of states
  - avoid revisiting states



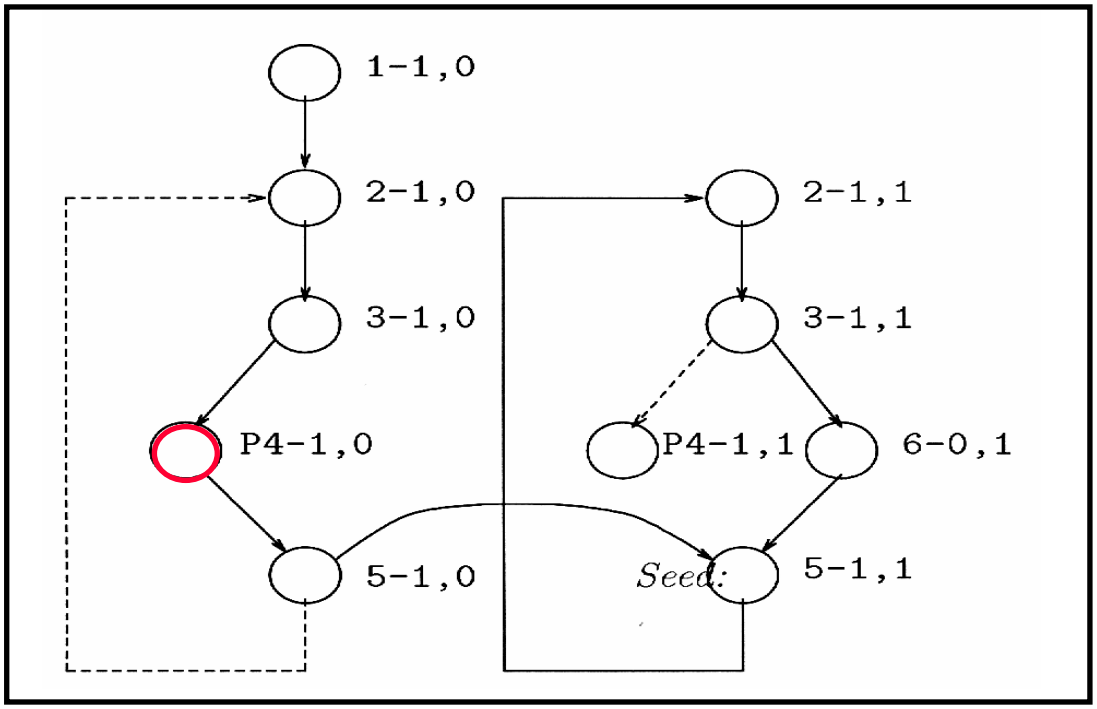
detecting non-progress

- prove absence or existence of non-progress cycles

progress state p



the nested depth-first search



- memory overhead: 2 bits per state
  - Tarjan's dfs requires 2x32 bits per state
- worst-case time overhead: 2x dfs
- compatible with bitstate storage / hash-compact, etc.

state storage options

- exhaustive storage
  - very fast, but consumes most memory
- lossless one-way compression
  - huffman compression, runlength encoding
  - hierarchical indexing (Collapse)
- lossy compression
  - bitstate hashing (compress down to 2 bit positions/state)
  - hash-compact (compress down to 40 bits/state, stored in regular hashtable)
- minimized automaton representation
  - [HolzmannPuri98], STTT98, in Spin since 1/98

4. model checking in practice

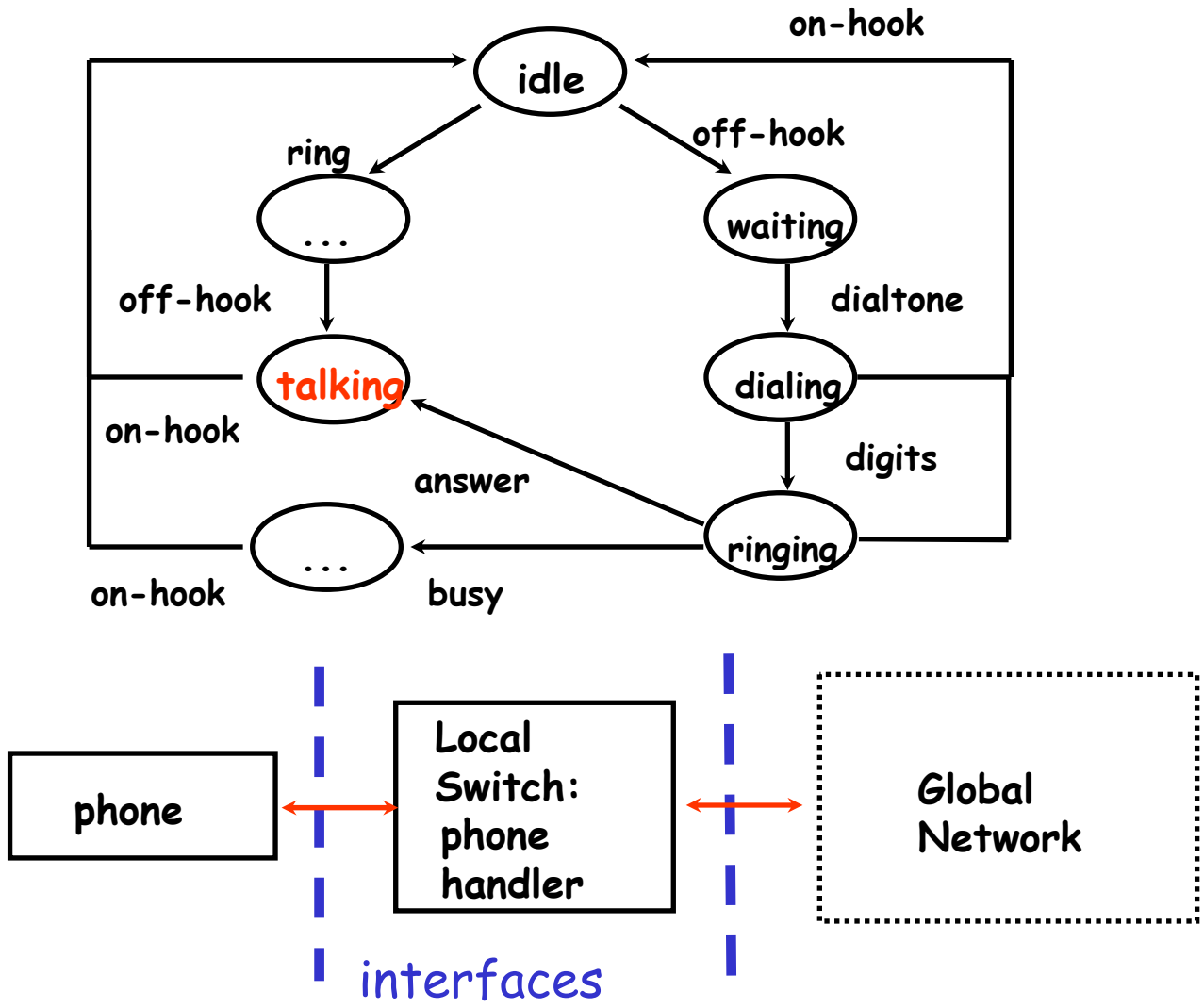
modeling and verification: the bottom line

- model checking is an exercise in abstraction: formalizing the interfaces between modules, without formalizing the details of the modules themselves

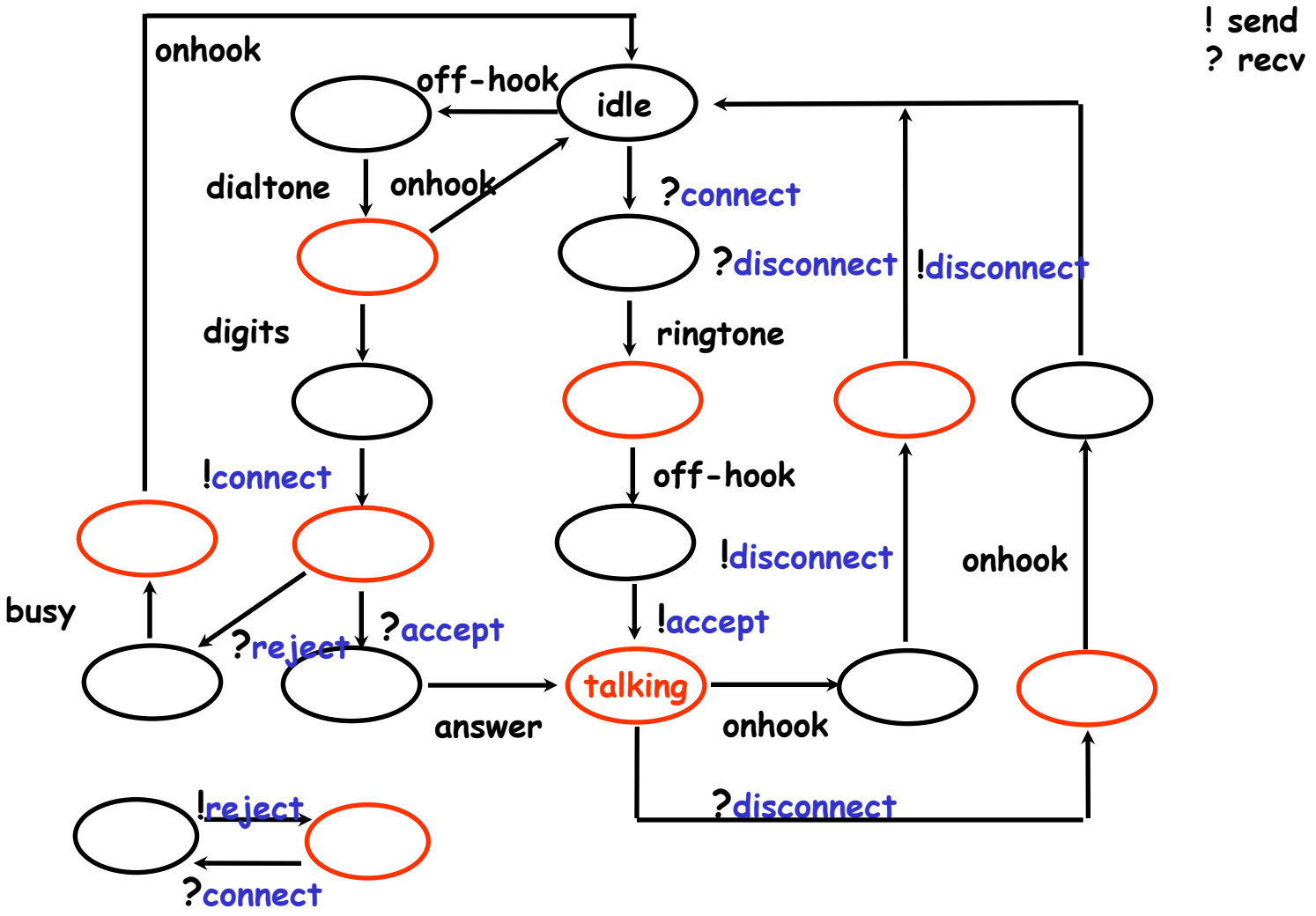
- the right abstraction can make seemingly intractable problems tractable

- Spin doesn't verify implementations, but it can thoroughly check a range of interesting properties of bounded models

an example: pots (plain old telephone service)

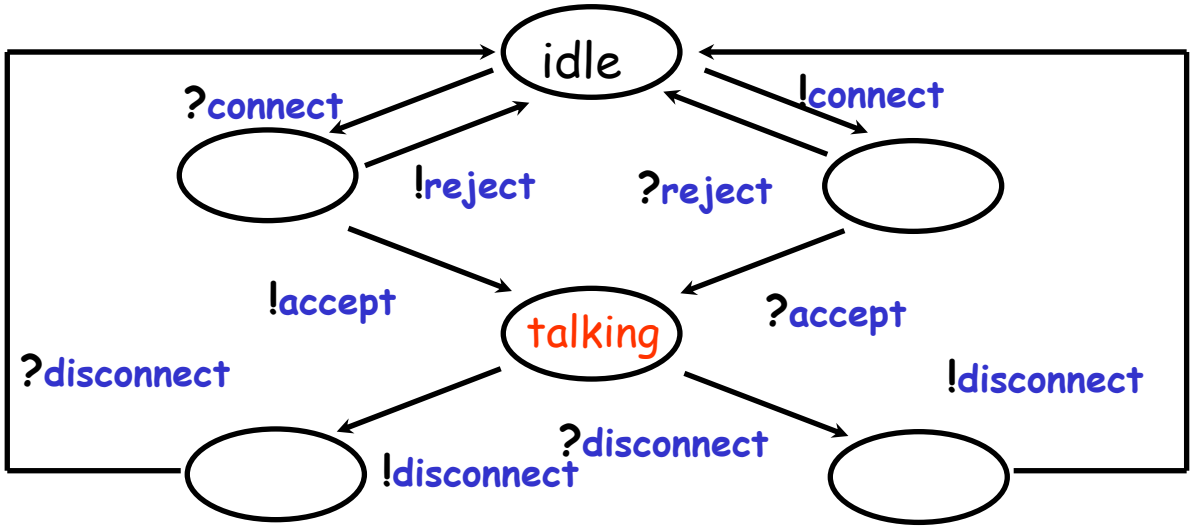


pots model (phone handler in switch)



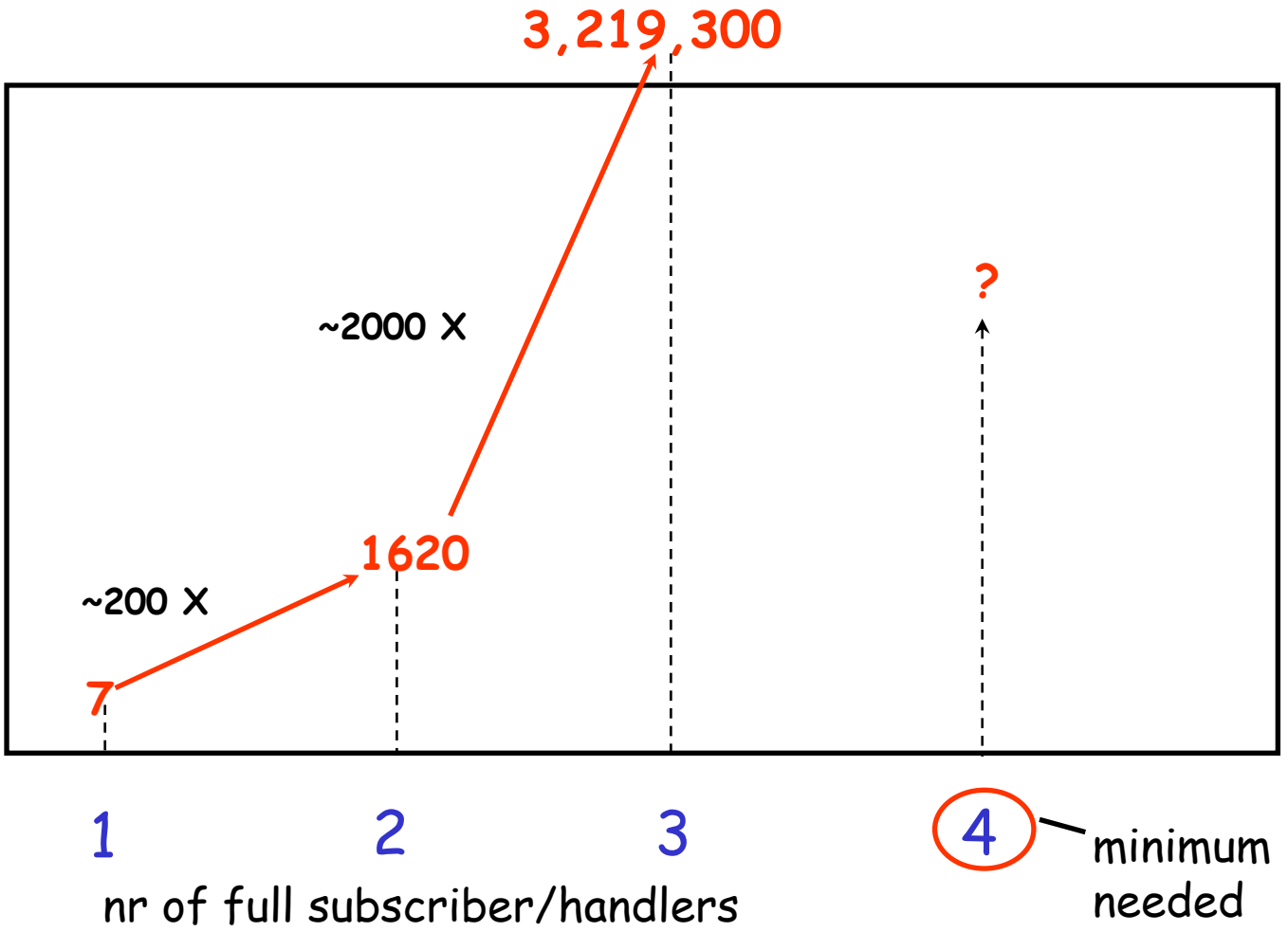
POTS subscriber x handler

- 40 x 113 = 4520 states per user
- with N users: $(4520)^N$ system states
- feasibility of model checking for 10, 100, 1000 subscribers?
- this model omits a crucial abstraction!
- abstraction of the network, as seen by each handler:

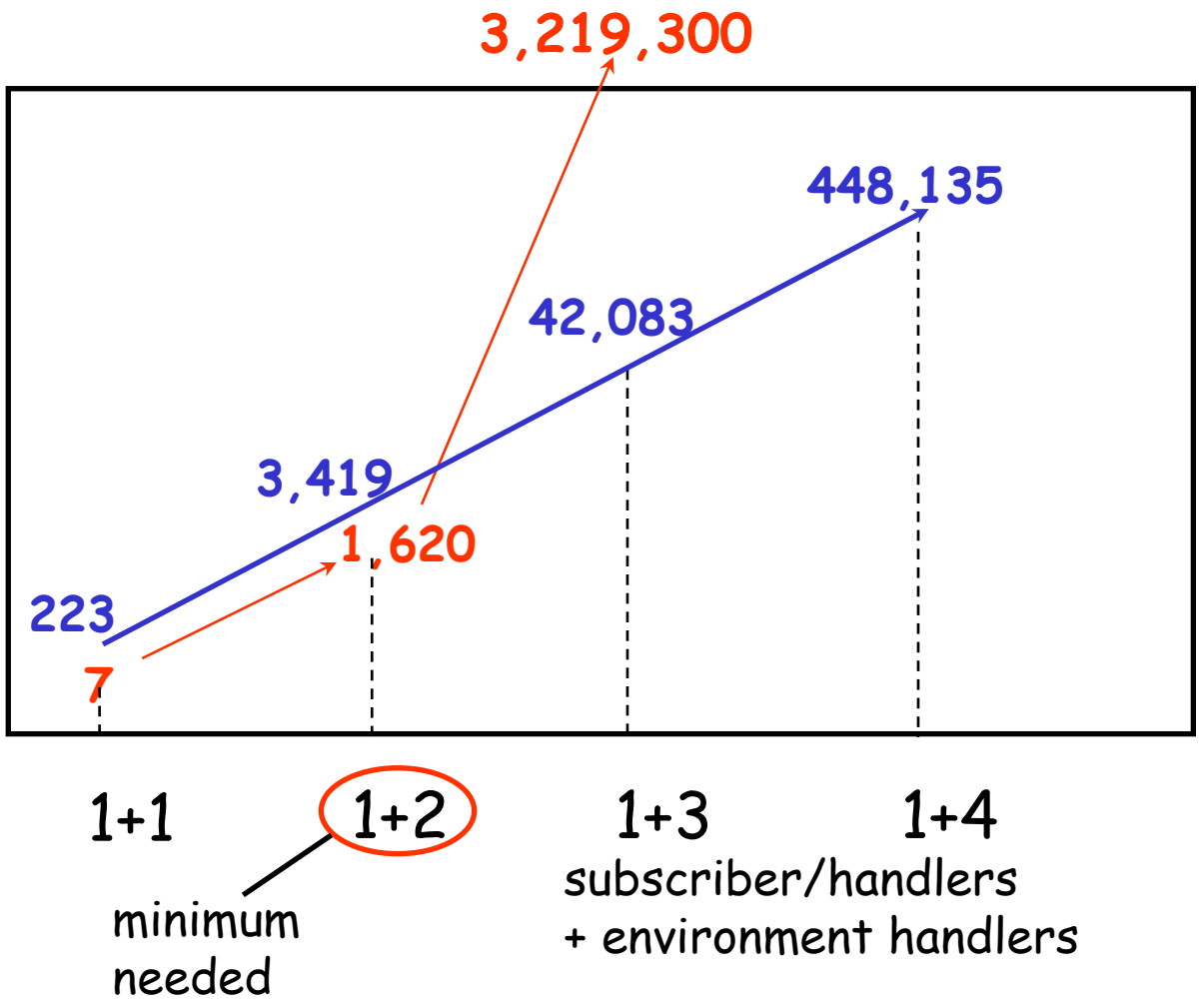


approx. 6 visible states

original model checking run
(reachable states, with p.o. reduction)



checking the revised model



model checking method when used for design verification

1. formalize all relevant design assumptions in an executable design abstraction (a model)
2. in a distributed system: processes define the elements of the model. the model details interface behavior (coordination) ***not*** internal behavior (computation)
3. make the design abstraction refutable by adding explicit and falsifiable requirements
4. the abstraction enables and empowers the analysis
5. the refinement of the model and of the properties proceed together -- start simple

reducing complexity

computational cost revisted

verification of ω–properties:

- property automaton of N states:
- R nr reachable states, S nr bytes used per state
- cost is roughly: 2xRxNxS time and RxNxS in memory

reduction strategies:

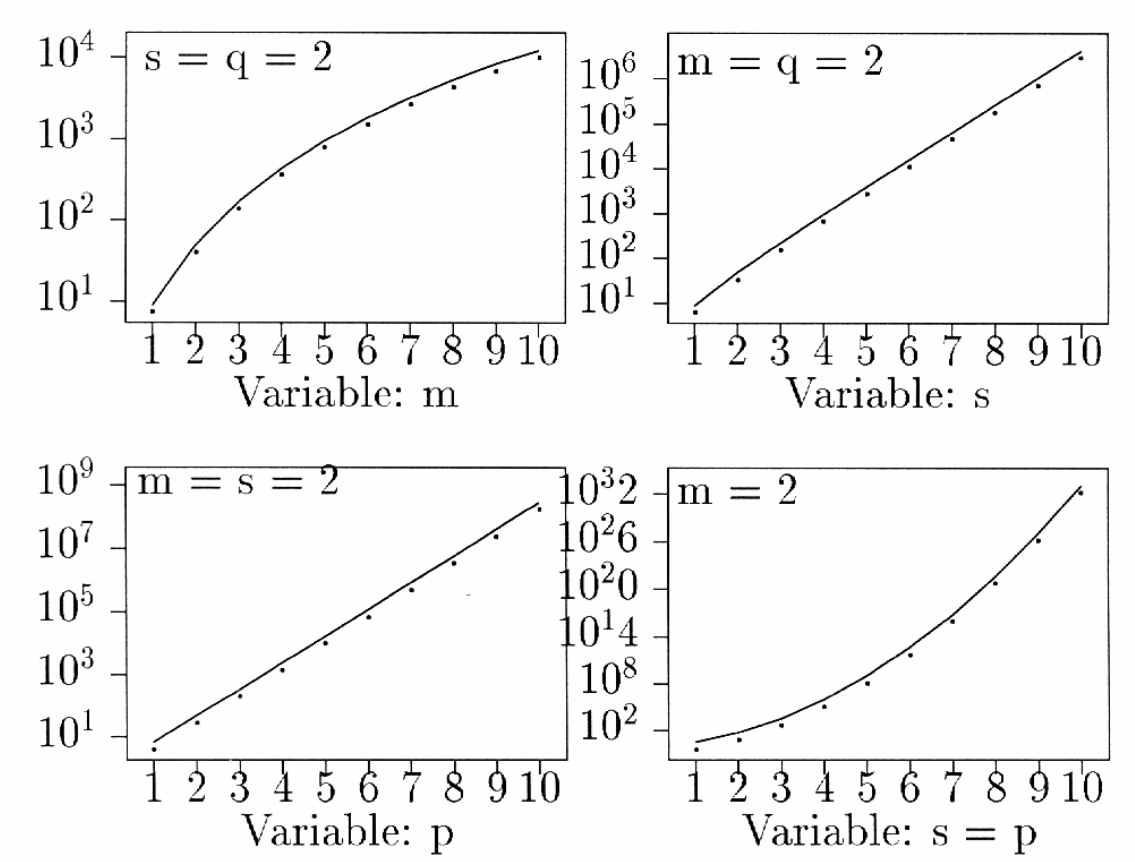
- try to reduce the size of N:
  - avoid LTL, or use a simpler property (separable properties)
- try to reduce the size of R:
  - model reduction (abstraction), symmetry reduction, etc.
  - partial order reduction (default)
  - state space caching, etc.
- try to reduce the size of S:
  - use compression (-DCOLLAPSE)
  - compute a DFA recognizer for RxS (-DMA=N)

reducing the size of R

- example: FIFO queues (buffers)
  - q = number of buffers
  - s = number of slots per buffer
  - m = number of different tokens that can be stored into the buffer
- how many unique states can this set of bounded data objects be in?

answer: $$R_Q = (\sum_{i=0}^{s} m^i)^q.$$

exponential effects



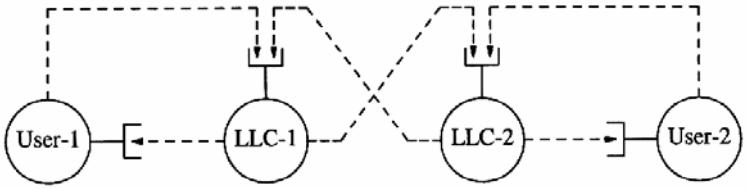
- q = nr of message channels
- s = capacity of each channel
- m = nr of distinct messages

model reduction: abstraction

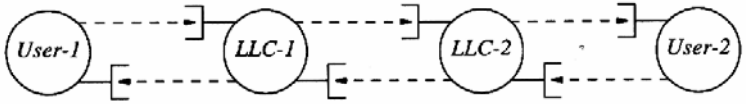
- exponential effects work both ways: up and down
- they can make simple problems computationally intractable
- or they can make hard problems solvable if you know which parameters to tune….
- if the problem is too complex:
  - there is no free lunch: you have to find the right design abstraction
  - be suspicious of variables, counters, integer data items
  - think of how you would explain the principles of the design on a blackboard to a friend
  - use: Generalization, Abstraction, Modularity, Structure, Symmetry, etc. etc.

example: modeling IEEE 802.2 LLC

- two equivalent models of IEEE 802.2 Logical Link Control Protocol were constructed (~1987) for verification



1,851,049 unique reachable states



19,407 unique reachable states

why generalization works

- to prove p, Spin tries to find at least one example where !p holds
- to disprove absence of acceptance or non-progress cycles, it suffices to prove that at least one such cycle exists
- it is not necessary to find all possible violations of correctness requirements
- let E be the set of runs of the model
- we must show that E contains at least one run that violates/satisfies a property

- *adding* execution sequences to E cannot ever cause us to miss errors (it preserves all existing errors)

reduction and abstraction

- we can add runs to a model without affecting the outcome of a positive verification
- consider a model of a phone:
  - off-hook, on-hook, dial digits, flash, etc. can only happen in a specific order
  - generalization: generate off-hook, on-hook, digits, flashes, etc. randomly, with a one-state model...
  - all runs of the detailed model are contained in the one-state model -- yet the second model is more tractable than the first.
- a less detailed model is often more tractable, and often allows for an even stronger verification result
- it does introduce the possibility of a false negative (i.e., it is fail-safe) -- which if present would require a softening of the generalization

example: the design of a server

- consider a server for q users, with s slots reserved per user and just one type of message per user
- which is better:
  - 1 shared server queue for all users, with qxs slots, or
  - q separate server queues, 1 per user, each with s slots ?

calculate:

$$\sum_{i=0}^{q \times s} q^i \quad \textit{versus} \quad (s+1)^q$$

---

the difference:

For s=5, q=3:

$$\sum_{i=0}^{q \times s} q^i = \sum_{i=0}^{15} 3^i = 21,523,360$$

$$(s+1)^q = 6^3 = 216$$

partial order reduction

- some types of reduction can be automated inside the model checker
- the global reachability graph contains many paths (runs) that are equivalent for given requirements
- some cases of equivalence can be automatically detected and avoided
- specifically, this applies to complexity introduced by the non-determinism of concurrent process execution (asynchronous interleaving)

definition of independence

- two actions are independent at state S if
  - both are enabled at S
  - the execution of one cannot disable the other
  - the combined execution of both has the same effect on S, independent of the order of execution
- strong independence
  - two actions are strongly independent if they are indepenent at every reachable state where both are enabled
- safety (a static property…)
  - an action is safe if it is strongly independent from all other actions in the system
  - an action is conditionally safe for condition *C* if it is safe when *C* holds

Peled's reduction theorem

S = reachable system state
T = set of all executable transitions at S

Divide T into 3 disjoint subsets:

- Dis(S) (transitions that cannot be executed from S)
- Sel(S) (enabled transitions, selected in reduction)
- Ign(S) (enabled transitions, but not selected)

THEOREM
It is sufficient for the preservation of SAFETY and LIVENESS properties to explore only the transitions from Sel(S) provided that 3 conditions are satisfied:

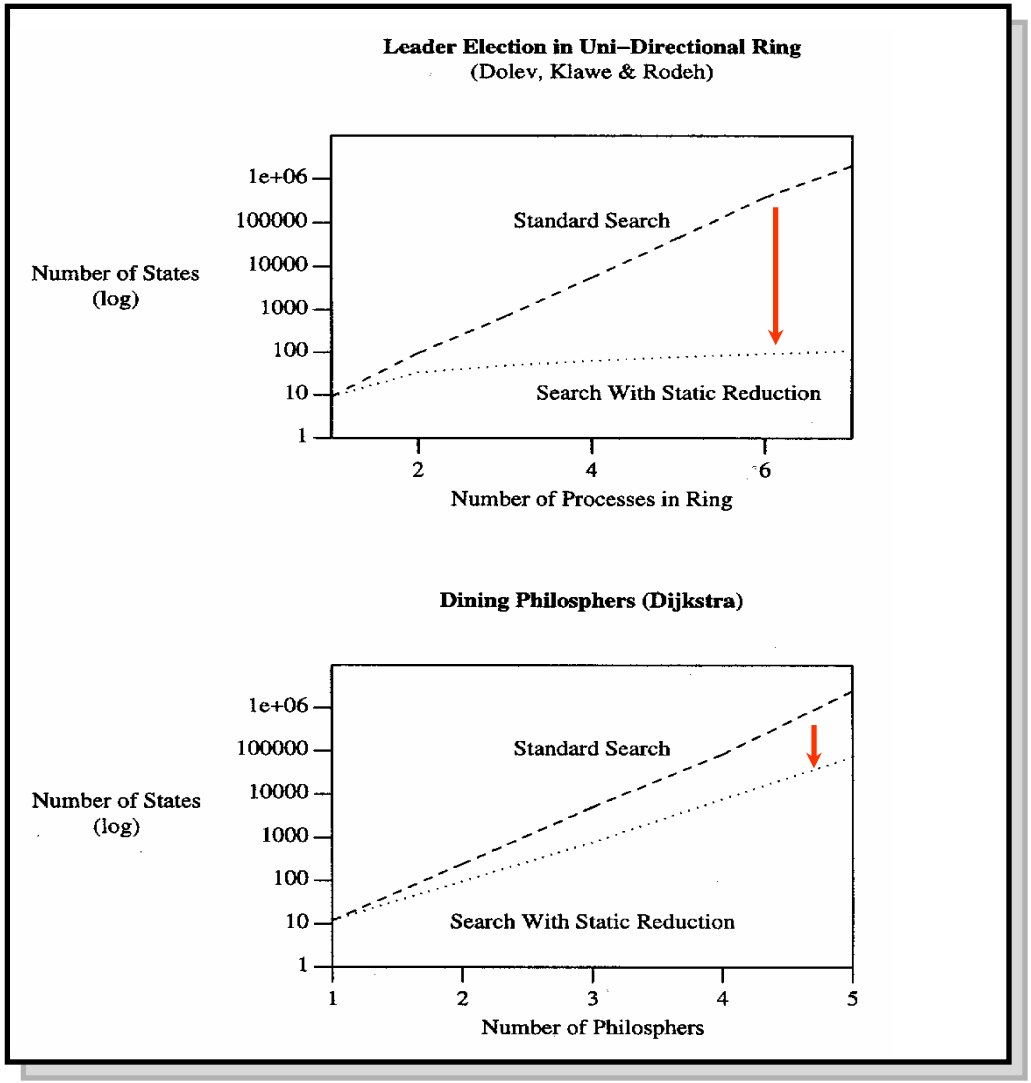
C1
No execution sequence starting from S can exist in which a transition that
- is outside set Sel(S), and that
- is dependent on at least one transition in Sel(S)

becomes enabled without at least one transition from Sel(S) being executed first in that sequence.

C2
If Ign(S) is non-empty then no transition in Sel(S) can close a cycle (i.e., return the search to a state that is on the depth-first search stack at S).

C3
The execution of none of the transitions in set Sel(S) touches an object that is visible to the property being checked (i.e., can alter the truth value of a proposition in the Buchi automaton, or the LTL property checked).

effect of partial order reduction

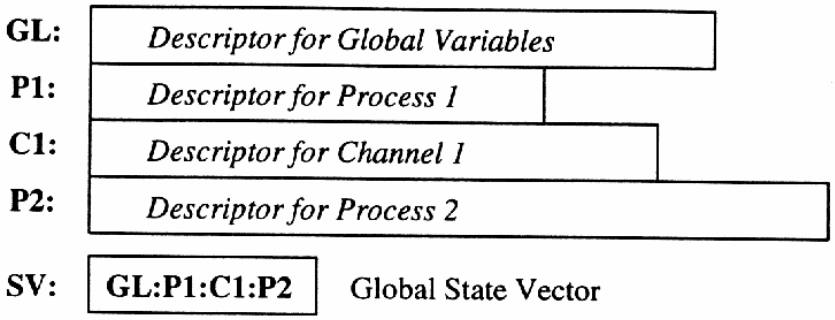


reducing the size of S: state compression

- the default compression in SPIN omits fields from the state vector that hold redundant data (padding, rendez-vous channels, etc.)
- static Huffman coding can reduce the size of a state by 60-70% more, but adds a run-time penalty of approx. 300% [PSTV92]
- state space caching methods are not well-behaved
- index-compression (-DCOLLAPSE) reduces states by 60-70% more, and adds a 10-20% runtime penalty
- DFA-compression (-DMA=N) reduces memory used by 10x-100x but adds a 10x-100x runtime penalty

example of index compression

The Main Components of the State Vector Are Separated and Numbered. Only the Index Numbers are Stored in the Global State Vector.



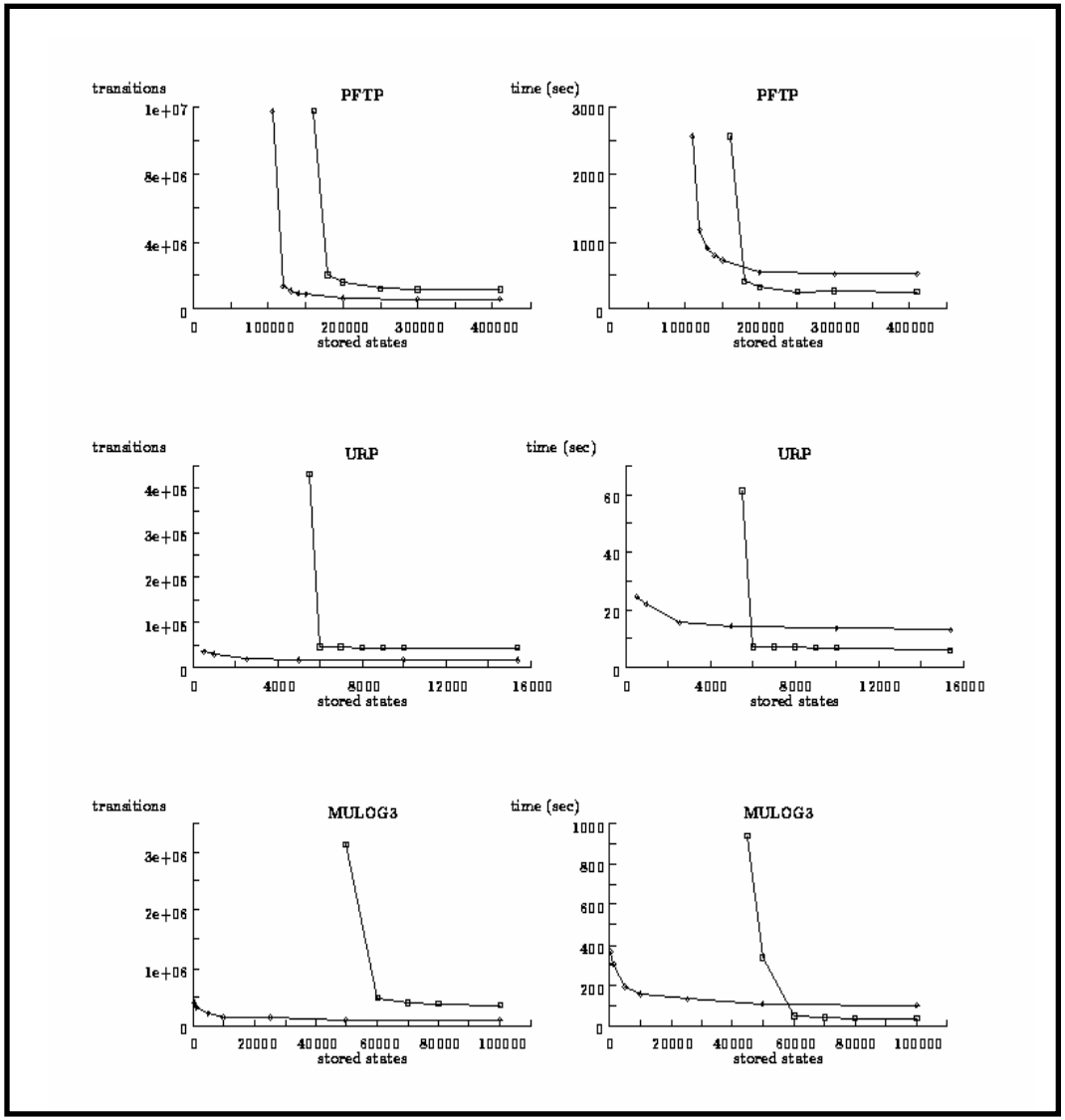
Effect of Compression

Type of Run	#States	Memory (Mb)	Time (sec.)
Standard	2,435,220	156.59	107.56
Compressed	2,435,220	59.57	123.46

(Go-Back-N Sliding Window Protocol)

state space caching [H85, HG92]

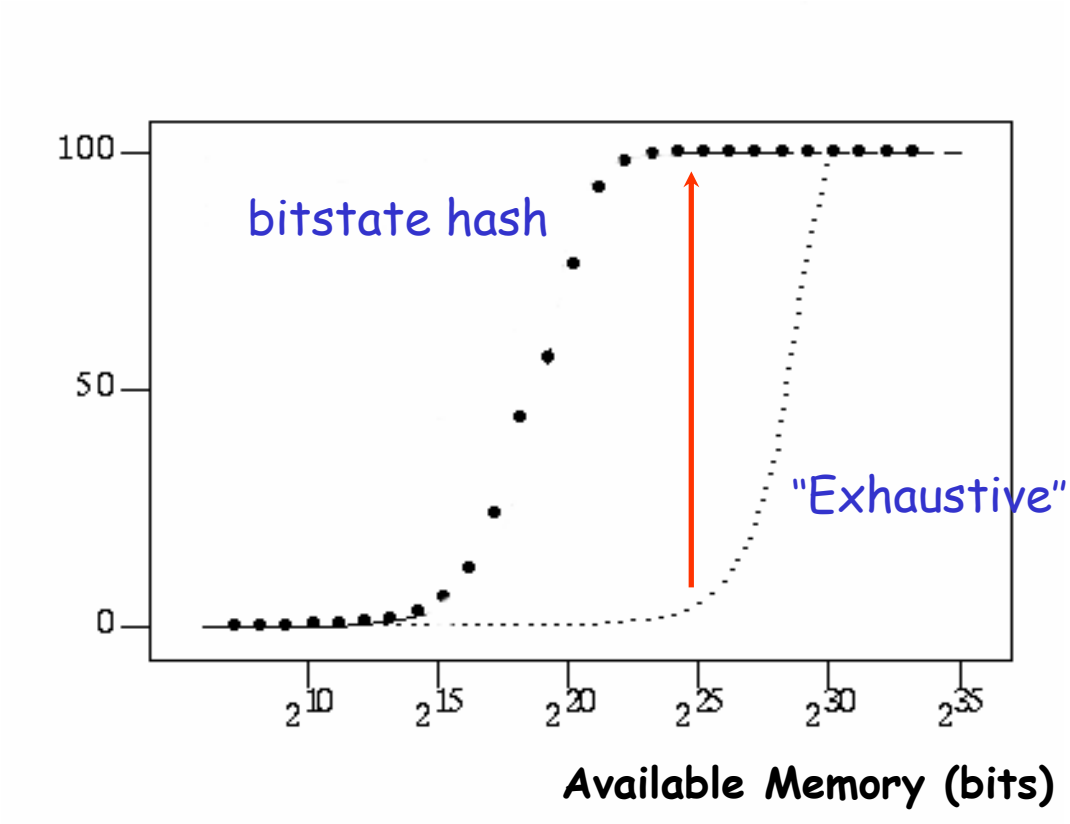
State Caching methods are not well-behaved:



bit-state hashing (1987)

- not lossless -- only useful as a last resort
- uses CRC-polynomials to compute checksums of state descriptors
- the checksum values are used as bit-addresses in a large bit-array in memory
- effectively this stores about. 28..30 bits of information in 1 single bit of memory….
- runs extremely fast (replaces state comparison and copying with CRC computations), and can significantly increase search coverage for large problems

effect of bitstate hashing



(Data: Data Transfer Protocol)

in numbers

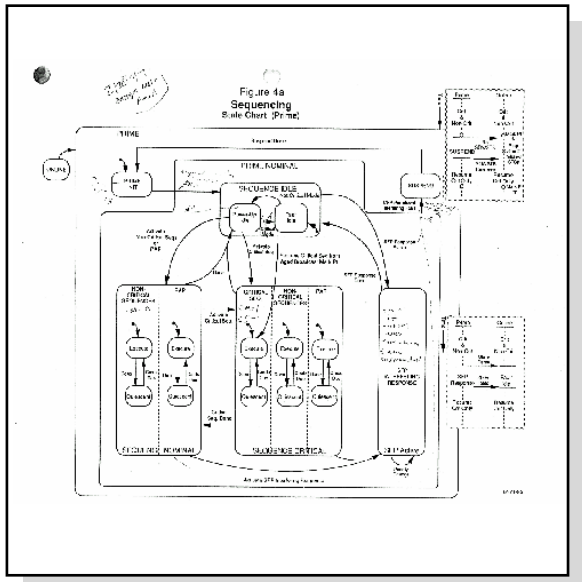
- with 64 Mbyte of available memory
- and 64 bytes per reachable state
- and 64 Million reachable states...
  - exhaustive search runs exhausts memory at 1 million states; the effective coverage is 1.5%
  - bitstate hashing can store up to 8x64 = 512 Million states -> it can realize 100% coverage...
  - but....., it cannot *guarantee* this
  - bitstate hashing can increase the confidence when exhaustive proof is impossible

two nasa applications

- Generic dual control software verification
  - completed: August 1997
- Deep Space 1, verification of new controller software
  - to be launched: 22 October 1998
  - flyby of Mars, a comet, and an asteroid

mark-rollback procedures

- a dual controller design
  - verification of mark-rollback procedure
- Spin model checking effort:
  - 3 people, 1 from NASA, 1 from JPL, 1 from Bell Labs
    - 4 weeks to build model (642 lines in Spin)
    - 1 week to perform and document the model checking
    - approx. 150K states, 4 seconds/run
    - 3 serious design errors found



requirements checked

‘‘When a fault is seen by an SFP monitor in the prime a response is triggered by the SFP manager. The SFP manager in the prime deactivates all sequences when it sees a response request flag for a response which is enabled and freezes the mark-point aging function. The SFP manager sets the appropriate STM flags to show that the SFP is active and that a response will be executed. These flags are updated in the STM at the end of the current second. An STP fault triggered in second N+1 becomes visible in the STB in RTI-5 of second N+2.’’

Translation:

If a fault occurs during a critical sequence, both Prime and Online system must roll-back to the last valid mark-point, and resume operation.

In Propositional Temporal Logic:

[] (p -> <> q)

where:

p = (Tprime < 3) && SFP && !CS && CM && (Mark == 6)

q = (PC == 1) && !SFP && CS && CM

loss of sync on roll-back

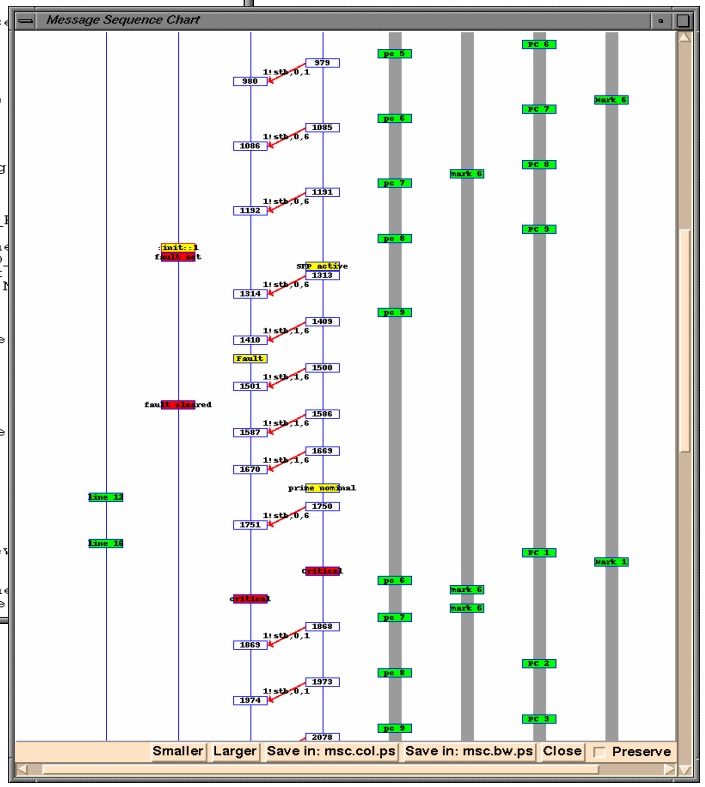
SPIN CONTROL 3.0.0 -- 10 May 1997 -- File: spec5

File.. Help Simulate.. Verify.. LTL.. Fsm view.. Line#: Find:

```
SEQUENCE_CRITICAL:
        printf("MSC: ~R Critical\n");
        /* This GDRS is prime */
        /* t is prime 125 ms clock */
        do
        ::
         if
          ::RTI_0-> SfpFault = globSfpFault
          ::RTI_1-> CSSV[PID] = 16;
        /*          printf("MSC: 16 PC SFP Mark %d %d %d  \n", PC, SFP, Mark) */
          ::RTI_2->
          ::RTI_3->
                atomic{
                        if
                        :: SfpFault ->
                                SFP = 1;
                                CS = 0;
                                goto SFP_Active /* SFP Respons
                        :: !SfpFault
                        fi
                }
         ::RTI_4->
         ::RTI_5->
                STB!stb(STM[1], STM[2]);  /* always bc SFP = 0
         ::RTI_6->
                atomic{
         /* online in ONLINE_SEQUENCE_CRITICAL CSSV 10    */
         /* looks for prime to be in SEQUENCE_CRITICAL state,
         /* if it is it sets SynchronousStart to TRUE clearing
         /* this string to start its copy of the cs in synch
                if
                :: (SynchronousStart) ->
                        if /* age the old Mark Point -not END_
                        ::(cseq[PC] == END_PROG)->
                                goto aPower_up_idle     /* Don
                        ::((cseq[PC] < MARK)&&(cseq[PC] != END_
                                /* this location is not a mark
                                buf[3] = buf[2];   /* move old
                                buf[2] = buf[1];
                                buf[1] = buf[0];
                                buf[0] = Mark;
#if defined(LTLb) || defined(LTLc) || defined(LTLc2) || define
defined(LTLd2)
                                PM = buf[3];
#endif
#ifdef LTLa
                                PrimeBuffer = buf[3];
#endif
#if defined(LTLc) || defined(LTLc2) || defined(LTLd) || define
                                if
                                :: Tprime < 3 -> Tprime++
                                :: else
                                fi;
#endif
                                PC++; assert(PC < 14)
                        ::(cseq[PC] >= MARK)->  /* new MP
                                buf[3] = buf[2];
                                buf[2] = buf[1]; /* get the ne
                                buf[1] = buf[0];
                                buf[0] = PC;
                                Mark = PC;          /* record n
#if defined(LTLb) || defined(LTLc) || defined(LTLc2) || define
```

[] (p -> <> q)

counter-example



finding the error

the default Spin model checking run:

- S = 92 byte/state
- R = 123,718 states
- 219,009 transitions
- 1,312 error scenarios

computational expense:

- 11.698 Mb memory
- 4.96 CPU seconds time

squeezing memory:

- using a DFA recognizer for states:
  - 4.50 Mb
  - 39.08 seconds

DS1 -- deep space 1

- "new millenium remote agent" design
  - verification of a new experimental distributed controller design
  - maintaining database consistency
- 2 people from NASA/Ames Research Center
  - 8 weeks to build model (354 lines in Spin)
  - 1 week to run and document verification
  - approx. 300K states, 10 seconds/run
  - 4 critical properties checked, all failed:
  - 4 serious errors found

Developer's reaction:

> I thought that formal methods advocates wanted to `prove correctness' of software, which I believe is impossible.
>
> However, what you have been doing is finding places where the software violates design assumptions, which is different. You guys have discovered things that we almost certainly would never have caught any other way
>
> To me you have demonstrated the utility of this approach beyond any question.

some references

- [Sistla&Clark85] A.P. Sistla and E.M. Clarke, *The complexity of propositional linear temporal logic.* J.ACM, 32, 1985, 733-749.
  - *PSPACE-completeness result*
- [Vardi&Wolper86] M.Y. Vardi and P. Wolper, *An automata-theoretic approach to automatic program verification.* Proc. 1st IEEE Symp. on Logic in Computer Science. 1986. 322-331.
  - *Introduced model checking by language intersection.*
- [Holz97] G.J. Holzmann, *The model checker Spin.* IEEE Trans. on Software Eng. 23, 5, 1997, 279-295.
  - *Generic overview paper for Spin with extensive references.*
- [Holz98] G.J. Holzmann, *An analysis of bitstate hashing.* Formal Methods in Systems Design, 13, 3, 1998, 287-305.
  - *A review of the hashing method from 1987.*
- url: http://netlib.bell-labs.com/netlib/spin/whatispin.html