Advanced
SPIN Tutorial

Gerard Holzmann

NASA/JPL Laboratory for
Reliable Software
Pasadena, CA

email: gholzmann@acm.org
http://spinroot.com/gerard

Theo Ruys

University of Twente
The Netherlands

email: ruys@cs.utwente.nl
http://www.cs.utwente.nl/~ruys/

FM 2003 Tutorial Pisa, Italy, 8 September 2003

Common Design Flaws

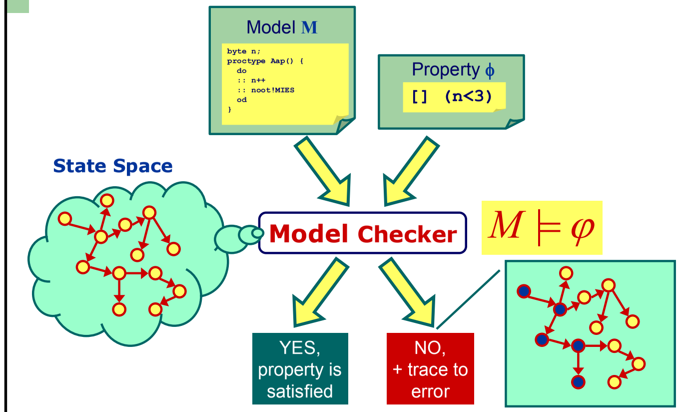
- Deadlock
- Livelock, starvation
- Underspecification
  - unexpected reception of messages
- Overspecification
  - Dead code
- Violations of constraints
  - Buffer overruns
  - Array bounds violations
- Assumptions about speed
  - Logical correctness vs. real-time performance

In designing distributed systems:
network applications,
data communication protocols,
multithreaded code,
client-server applications.

Designing concurrent (software) systems is so hard, that these flaws are mostly overlooked...

Fortunately, most of these design errors can be detected using model checking techniques!

Model Checking



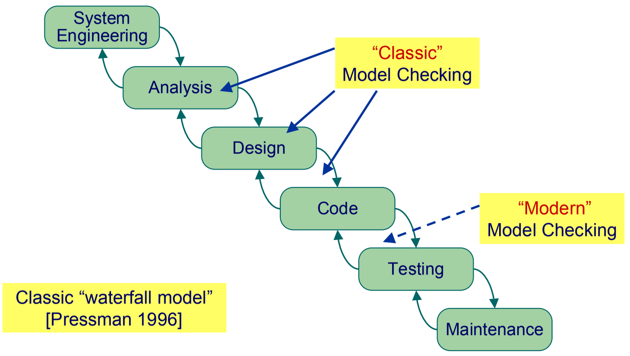
```
byte n;
proctype Aap() {
  do
  :: n++
  :: noot!MIES
  od
}
```

Verification vs. Debugging

- Two (extreme) approaches with respect to the application of model checkers.
  - verification approach: tries to ascertain the correctness of a detailed model M of the system under validation.
  - debugging approach: tries to find errors in a model M.
- Model checking is most effective in combination with the debugging approach.

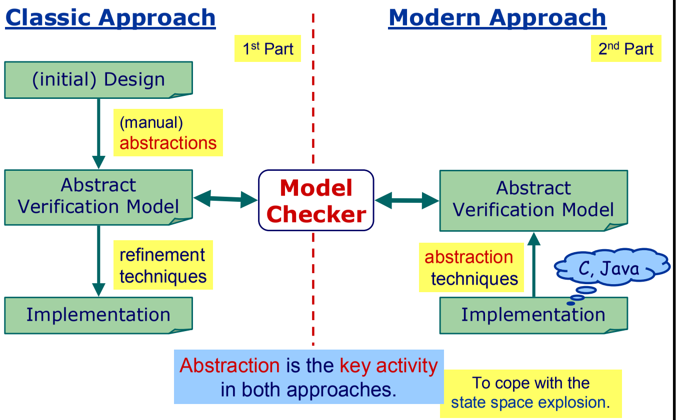
Automatic verification is *not* about proving correctness, but about finding bugs much earlier in the development of a system.

System Development



Classic "waterfall model" [Pressman 1996]

Classic vs Modern Approach



Abstraction is the key activity in both approaches.

To cope with the state space explosion.

Overview

Part 1

- Introduction
- Effective SPIN: the art of Promela Modelling
- Checking Invariance
- Systematic Verification
- Solving optimisation problems with SPIN 4.0

Part 2

- How SPIN works:
  - Some automata theory
- Complexity issues
  - Reduction and compression
- Model extraction
  - Software model checking

SPIN – Introduction

- Major versions:

1.0	Jan 1991	initial version [Holzmann 1991]
2.0	Jan 1995	partial order reduction
3.0	Apr 1997	minimised automaton representation
4.0	Jan 2003	embedding of C code + BFS

- Some success factors of SPIN
  - "press the button" verification (model checker)
  - very efficient implementation (using C)
  - nice graphical user interface (Xspin)
  - not just a research tool, but well supported
  - contains more than two decades research on advanced computer aided verification (many optimization algorithms)

Software System Awards

2001 Gerard Holzmann

1983 Unix
1986 TeX
1997 Tcl/Tk
2002 Java

Documentation on SPIN

- SPIN's starting page:
  http://spinroot.com
  - Basic SPIN manual
  - Getting started with Xspin
  - Getting started with SPIN
  - Examples and Exercises
  - Concise Promela Reference (by Rob Gerth)
  - Proceedings of all ten SPIN Workshops
- Gerard Holzmann's website for papers on SPIN:
  http://spinroot.com/gerard/
- SPIN version 1.0 is described in [Holzmann 1991].

(Still) available in PDF format from www.spinroot.com.

September 10th 2003!
Gerard J. Holzmann
The Spin Model Checker
Primer and Reference Manual
Addison Wesley
ISBN 0-32122-862-6, 608 pages.
describes SPIN up to version 4.0.

Promela Model

- A Promela model consist of:
  - type declarations
  - channel declarations
  - global variable declarations
  - process declarations
  - [`init` process]

`mtype`, constants,
typedefs (records)

`chan ch = [cap] of {type, ...}`
asynchronous: *cap* > 0
rendez-vous: *cap* == 0

- simple vars
- structured vars
- vars can be accessed by all processes

behaviour of the processes:
local variables + statements

initialises variables and
starts processes

Promela statements

most important
Promela statemens

are either executable
or blocked

`skip`	always executable
`assert(<expr>)`	always executable
expression	executable if not zero
assignment	always executable
`goto`	always executable
`if`	executable if at least one guard is executable
`do`	executable if at least one guard is executable
`break`	always executable (exits `do`-statement)
send `(ch!)`	executable if channel `ch` is not full
receive `(ch?)`	executable if channel `ch` is not empty
`atomic{ ... }`	executable if first statement is executable
`d_step{ ... }`	executable if first statement is executable
`timeout`	executable if no other statement is executable

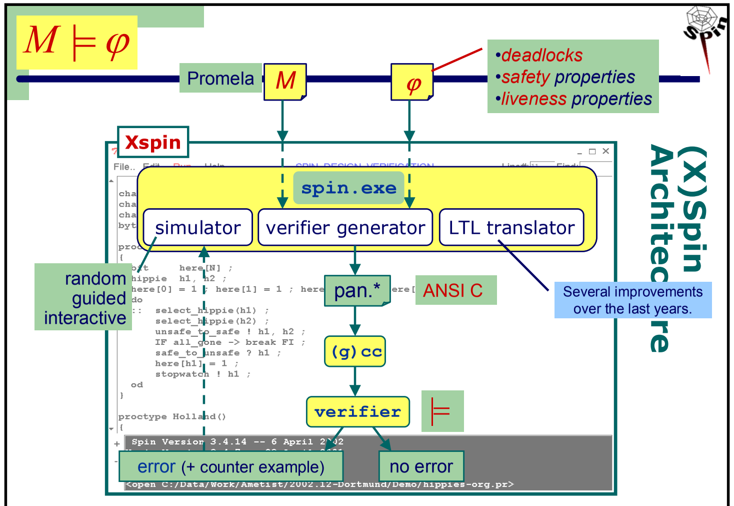
Basic recipe to check $M \models \varphi$

Properties:
1. deadlock
2. assertions
3. invariance
4. liveness (LTL)

1. Sanity check
   Interactive and random simulations.
2. Partial check
   Use SPIN's bitstate hashing mode to quickly sweep over the state space.
   states are not stored; fast method
3. Exhaustive check
   If this fails, SPIN supports several options to proceed:
   1. Compression (of state vector)
   2. Optimisations (SPIN-options or manually)
   3. Abstractions (manually, guided by SPIN's slicing algorithm)
   4. (Bitstate hashing)

(X)Spin Architecture

$M \models \varphi$



- *deadlocks*
- *safety properties*
- *liveness properties*

random
guided
interactive

ANSI C

Several improvements over the last years.

```
bit      here[N] ;
hippie   h1, h2 ;
here[0] = 1 ; here[1] = 1 ; here
do
::  select_hippie(h1) ;
    select_hippie(h2) ;
    unsafe_to_safe ! h1, h2 ;
    IF all_gone -> break FI ;
    safe_to_unsafe ? h1 ;
    here[h1] = 1 ;
    stopwatch ! h1 ;
  od
}

proctype Holland()
{
Spin Version 3.4.14 -- 6 April 2002
<open C:/Data/Work/Ametist/2002.12-Dortmund/Demo/hippies-org.pr>
```

Simulation Algorithm

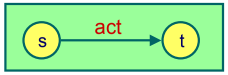
deadlock ≡ allBlocked

```
while (! error & ! allBlocked) do
    ActionList menu = getCurrentExecutableActions();
    allBlocked = (menu.size() == 0);
    if (! allBlocked)
        Action act = menu.chooseRandom();
        error = act.execute();
    fi
od
```

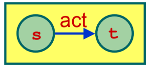
act is executed and the system enters a new state

interactive simulation: act is chosen by the user

Visit all processes and collect all executable actions .



Verification Algorithm (1)



- SPIN uses a depth first search algorithm (DFS) to generate and explore the complete state space.

```
procedure dfs(s: state)
    if error(s) then report error fi
    add s to Statespace
    foreach successor t of s do
        if t not in Statespace then dfs(t) fi
    od
end dfs
```

states are stored in a hash table

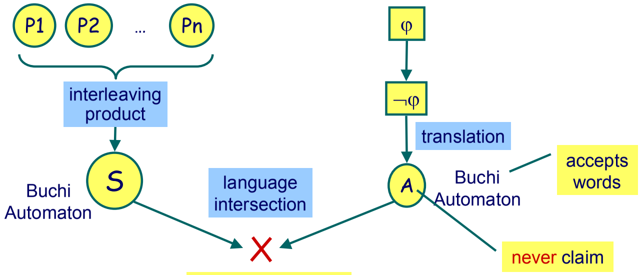
requires state matching

the old states s are stored on a stack, which corresponds with a complete execution path

Only works for safity properties.

- Note that the construction and error checking happens at the same time: SPIN is an on-the-fly model checker.

Verification Algorithm (2)



X should be empty

Procedure: search for an accepting state in the intersection, which is reachable from itself. In SPIN this is implemented by two basic DFS procedures. See [Holzmann et. al. 1996] for details.

Based on [Vardi & Wolper 1986]

Verification of Properties

safety property
- "nothing bad ever happens"
- invariance
  x is always less than 5
- deadlock freedom
  the system never reaches a state where no actions are possible

SPIN: find a trace leading to the "bad" thing. If there is not such a trace, the property is satisfied.

liveness property
- "something good will eventually happen"
- termination
  the system will eventually terminate
- response
  if action X occurs then eventually action Y will occur

SPIN: find a (infinite) loop in which the "good" thing does not happen. If there is not such a loop, the property is satisfied.

State vector

- A state vector is the information to uniquely identify a system state; it contains:
  - global variables
  - contents of the channels
  - for each process in the system:
    - ➤ local variables
    - ➤ process counter of the process
- It is important to minimise the size of the state vector.

state vector = m bytes
state space = n states

➡

storing the state space
may require n*m bytes

SPIN provides several algorithms to compress the state vector.

[Holzmann 1997 - State Compression]

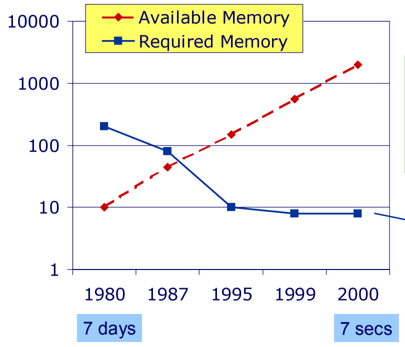
SPIN's Reduction Algorithms

- SPIN has several optimisation algorithms to make verification runs more effective:
  - partial order reduction
    idea: if in some global state, a process P can execute only "local" statements, then all other processes may be deferred until later
  - bitstate hashing (approximate)
    instead of storing each state explicitly, only one bit of memory is used to store a reachable state
  - hash compaction (approximate)
  - state vector compression ("zipping the individual states")
  - minimised automaton encoding of states (not in a hash table)
  - dataflow analysis: dead variable analysis, atomic merging
  - slicing algorithm ("give hints of what can be thrown away")

SPIN's power (and popularity!) is based on the these (default) optimisation/reduction algorithms.

Moore's Law & Advanced Algorithms

[Holzmann 2000 M'dorf]
- Verification results of `Tpc` (The phone company)



1980: pan
1987: bitstate hashing
1995: partial order reduction
1999: minimised automaton

memory requirements to (fully) verify **Tpc**

Effective Modelling

- BRP = Bounded Retransmission Protocol
  - alternating bit protocol with timers
  - 1997: exhaustive verification with SPIN and UPPAAL
  - 2002: optimised version of the original model
  - shows the effectiveness of a tuned model

	BRP 1997	BRP 2002
state vector	104 bytes	96 bytes
# states	1,799,340	169,208
Memory (Mb)	116.399	14.354

Both verified with SPIN 3.4.x

took upto an hour in 1997

took 2 sec. in 2002

Art of (Promela) Modelling

It's all about abstractions!

- expert user
  - uses 'assembler programming' approach to model building
  - knows how to exploit the directives and options of the model checker to optimise and tune the verification runs
  - realises that different versions of the model might be constructed for each different property
- space vs. time considerations, priorities:
  1. Number of states
  2. Size of the state vector
  3. Maximum search depth
  4. Verification time

Experimental science:
- several optimising options make it difficult to predict the behaviour of a certain verification run;
- many different ways to model a particular aspect
- use many controlled verification runs with different settings to conclude which modelling solution performs best.

Some Promela "Patterns"

[Ruys PhD Thesis 2001]

- Tool Support
- First Things First
- Macros
- Atomicity
- Randomness
- Bitvectors
- Subranges
- Abstract Data Types: Deque
- Lossy channels
- Multicast Protocols
- Reordering a Promela model
- Invariance

[Ruys SPIN 2003]

- Solving scheduling problems

Still in the pipeline...

- Modelling Time in Promela

Checking for "pure" atomicity

- Suppose we want to check that none of the atomic clauses in our model are ever blocked (i.e. pure atomicity).

1. Add a global bit variable:

```
bit aflag;
```

2. Change all atomic clauses to:

```
atomic {
  stat₁;
➡ aflag=1;
  stat₂;

  ...

  statₙ;
➡ aflag=0;
}
```

3. Check that `aflag` is always 0.

```
[]!aflag
```

e.g.

```
active process monitor {
  assert(!aflag);
}
```

Invariance

`[]P`

- `[]P` where `P` is a state property
  - safety property
  - invariance ≡ global universality or global absence [Dwyer et. al. 1999]:
    - 25% of the properties that are being checked with model checkers are invariance properties
    - BTW, 48% of the properties are response properties
  - examples:
    - `[] !aflag`
    - `[] mutex != 2`
- SPIN supports (at least) 7 ways to check for invariance.

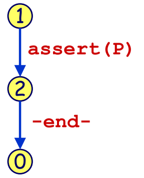
---

variant 1+2 - monitor process (single assert)

`[]P`

- proposed in older documentation on SPIN
- add the following monitor process to the Promela model:

```
active proctype monitor()
{
  assert(P);
}
```



- Two variations:
  - 1. monitor process is created first
  - 2. monitor process is created last

If the monitor process is created last, the -end- transition will be executable after executing `assert(P)`.

---

variant 3 - guarded monitor process

`[]P`

- Drawback of solution “1+2 monitor process” is that the `assert` statement is executable in every state.

```
active proctype monitor()
{
  assert(P) ;
}
```

```
active proctype monitor()
{
  atomic {
   !P -> assert(P) ;
  }
}
```

- The `atomic` statement only becomes executable when P itself is not true.

We are searching for a state where `P` is not true. If it does not exist, `[]P` is true.

---

variant 4 - monitor process (do assert)

`[]P`

- From an operational viewpoint, the following monitor process seems less effective:

```
active proctype monitor()
{
  do
  :: assert(P)
  od
}
```

- But the number of states is clearly advantageous.

variant 5 - never claim (`do assert`)

[]P

- also proposed in SPIN's documentation

```
never {
  do
  :: assert(P)
  od
}
```

SPIN will synchronise the never claim automaton with the automaton of the system. SPIN also uses never claims to verify LTL formulae.

… but SPIN will issue the following unnerving warning:

warning: for p.o. reduction to be valid the never claim must be stutter-closed (never claims generated from LTL formulae are stutter-closed)

… and this never claim has not been generated…

variant 6 - LTL property

[]P

- The logical way…
- SPIN translates the LTL formula into an accepting `never` claim.

```
never { ![]P
TO_init:
  if
  :: (!P) -> goto accept_all
  :: (1)  -> goto TO_init
  fi;
accept_all:
  skip
}
```

variant 7 - `unless {!P -> ...}`

[]P

- Enclose the ***body*** of (at least) one of the processes into the following `unless` clause:

`{ body } unless { atomic { !P -> assert(P) ; } }`

- Discussion
  - \+ no extra process is needed: saves 4 bytes in state vector
  - \+ local variables can be used in the property P
  - – definition of the process has to be changed
  - – the `unless` construct can reach inside `atomic` clauses
  - – partial order reduction may be invalid if rendez-vous communication is used within body
  - – the ***body*** is not allowed to end

This is quite restrictive.

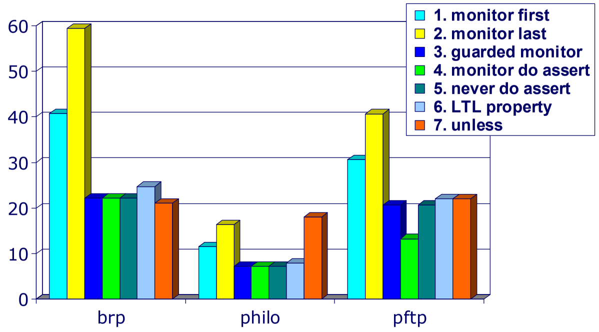
Note: disabling partial reduction (`-DNOREDUCE`) may have severe negative consequences on the effectiveness of the verification run.

Invariance experiments (1)

PII 300Mhz
128 Mb
SPIN 3.3.10
Linux 2.2.12

`-DNOREDUCE`
memory (Mb)

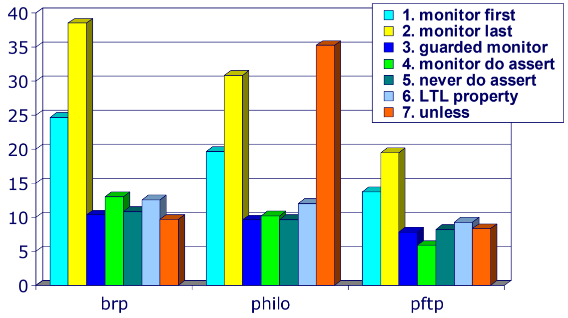
NO partial order reduction



Invariance experiments (2)

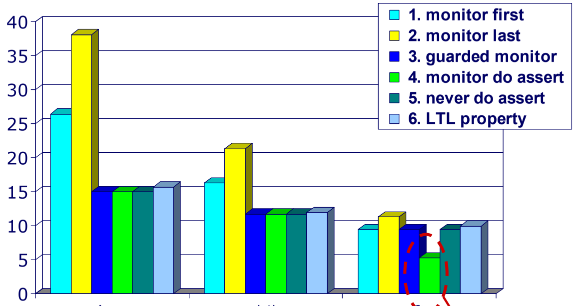
`-DNOREDUCE` — NO partial order reduction

time (sec)



Invariance experiments (3)

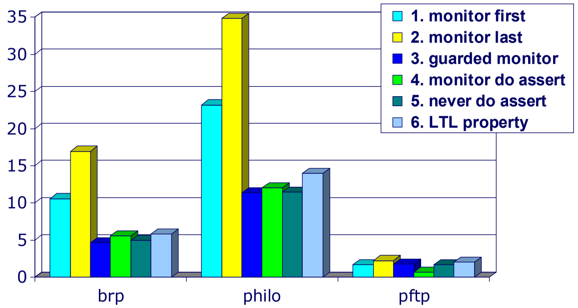
default settings
memory (Mb)



seems attractive…

Invariance experiments (4)

default settings
time (sec)



Invariance - Conclusions

- The methods 1 and 2 "monitor process with single `assert`" performed worst on all experiments.
  - When checking invariance, these methods should be avoided.
- Variant 4 "monitor do assert" seems attractive, after verifying the `pftp` model.
  - unfortunately, this method modifies the original `pftp` model!
  - the `pftp` model contains a `timeout` statement
  - because the `do-assert` loop is always executable, the `timeout` will never become executable

  ⇒ never use variant 4 in the presence of `timeout`s
- Variant 3 "guarded monitor process" is the most effective and reliable method for checking invariance.

Invariance - Conclusions

- Generalizing, if one need to check

  `[] (P && Q && ... && Z)`

  one should use:

```
active process monitor()
{
  if
  ::  atomic {!P -> assert(P)}
  ::  atomic {!Q -> assert(Q)}
  ::  ...
  ::  atomic {!Z -> assert(Z)}
  fi
}
```

Data Space Explosion (1)

- Industrial size verification projects do not only suffer from the infamous state space explosion, but also suffer from a “data space explosion”.
- Management of information and data
  - Many documents (specifications) from many parties
  - Several versions of the same document
  - Consecutive versions of validation models
  - Results of validation runs
- Annotations to the model are important:
  - identifying the source of information
  - discussing and explaining
    - modelling choices
    - abstractions
  - identifying points of attention

LP works like javadoc, but then in separate source files.

Use literate programming tools to annotate the (Promela) models. From a single source file one can either generate
- the plain Promela model or
- a nicely annotated LaTeX/HTML document.

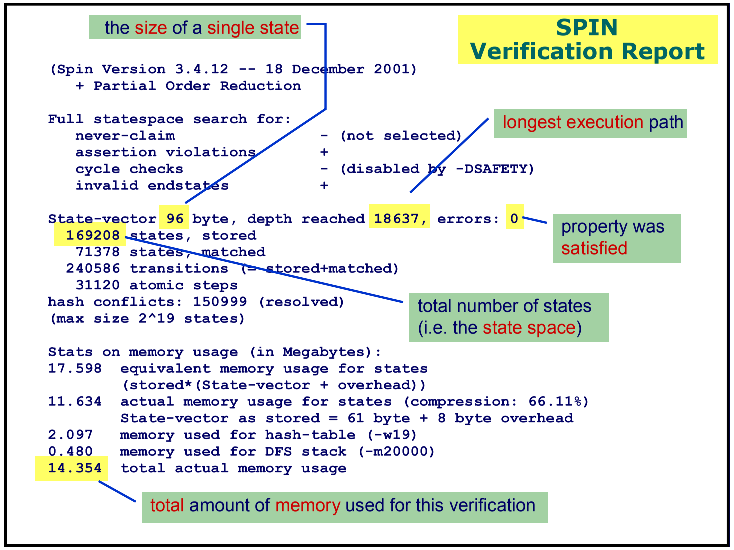
[Ruys & Brinksma 1998]

Data Space Explosion (2)

- Version space explosion of verification phase:
  - various models
    - variants: M_i
    - revisions: $M_{i,j}$
  - various properties: φ_i
  - validation results:
    - simulation traces
    - verification results
  - directives and options to build verifiers
  - notes and remarks on validation runs

Use Software Configuration Management (SCM) tools or version-control systems to save all verification data.

- Two important principles of verification phase:
  - Reverification in case of an error.
  - Validation results should be reproducible.
    - general engineering practice: use logbook



SPIN Verification Report

the size of a single state

```
(Spin Version 3.4.12 -- 18 December 2001)
   + Partial Order Reduction

Full statespace search for:
   never-claim             - (not selected)
   assertion violations    +
   cycle checks            - (disabled by -DSAFETY)
   invalid endstates       +

State-vector 96 byte, depth reached 18637, errors: 0
  169208 states, stored
   71378 states, matched
  240586 transitions (= stored+matched)
   31120 atomic steps
hash conflicts: 150999 (resolved)
(max size 2^19 states)

Stats on memory usage (in Megabytes):
17.598   equivalent memory usage for states
         (stored*(State-vector + overhead))
11.634   actual memory usage for states (compression: 66.11%)
         State-vector as stored = 61 byte + 8 byte overhead
2.097    memory used for hash-table (-w19)
0.480    memory used for DFS stack (-m20000)
14.354   total actual memory usage
```

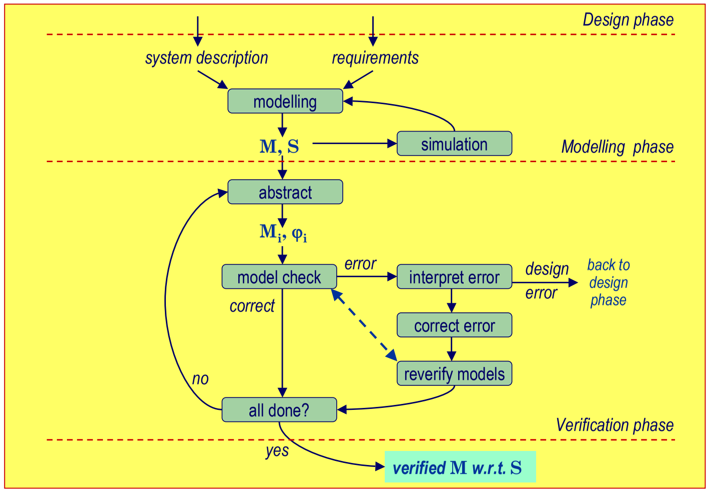
longest execution path

property was satisfied

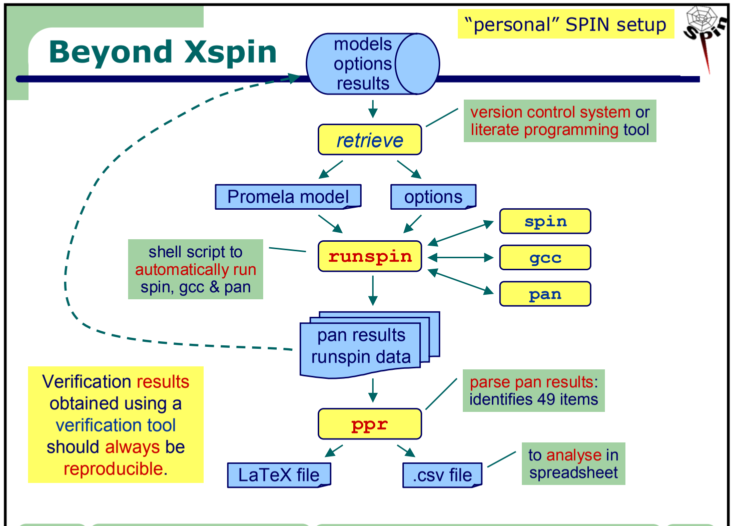
total number of states (i.e. the state space)

total amount of memory used for this verification

Systematic Verification Model



Beyond Xspin



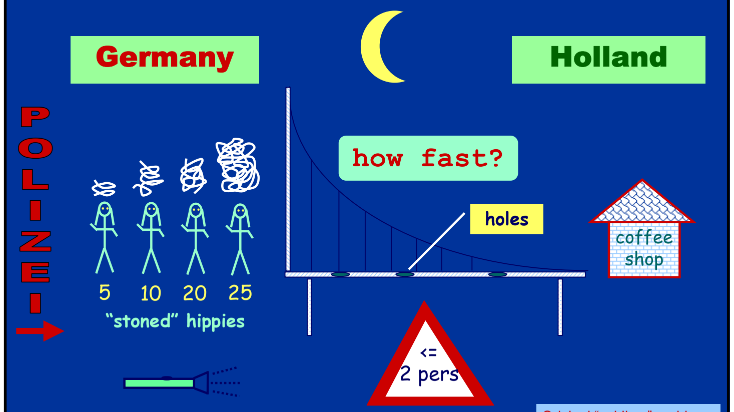
Verification results obtained using a verification tool should always be reproducible.

Scheduling with SPIN

- Introduction - Hippies problem
- SPIN 4.0 - new features
- Branch & Bound with SPIN 4.0
- Reordering the Promela model

See [Ruys SPIN 2003] for other examples:
- Traveling Salesman Problem
- A Job-shop scheduling problem

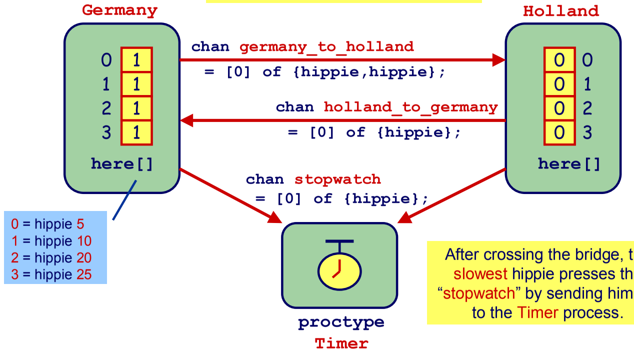
"Hippies" problem



Original "soldiers" problem: [Ruys & Brinksma 1998]

Hippies problem in Promela (1)

It is clear that we need to send two hippies over to Holland but only one hippie back with the flashlight.



proctype Germany

proctype Holland

`chan germany_to_holland = [0] of {hippie,hippie};`

`chan holland_to_germany = [0] of {hippie};`

`chan stopwatch = [0] of {hippie};`

proctype Timer

0 = hippie 5
1 = hippie 10
2 = hippie 20
3 = hippie 25

After crossing the bridge, the slowest hippie presses the "stopwatch" by sending himself to the Timer process.

Hippies problem in Promela (2)

```
chan germany_to_holland = [0] of {hippie, hippie};
chan holland_to_germany = [0] of {hippie};
chan stopwatch          = [0] of {hippie};
byte time;
...
active proctype Germany()
{
  bit      here[N];
  hippie   h1, h2;
  here[0]=1; here[1]=1; here[2]=1; here[3]=1;
  do
  ::  select_hippie(h1);
      select_hippie(h2);
      germany_to_holland ! h1, h2;
      IF all_gone -> break FI;
      holland_to_germany ? h1;
      here[h1] = 1;
      stopwatch ! h1;
  od
}
```

```
#define hippie byte
```

A hippie is randomly chosen from the hippies that are still "here".

```
#define all_gone \
  (here[0]+here[1]+ \
   here[2]+here[3]==0)
```

It can be modelled more effectively using bitvectors. See [Ruys PhD Thesis 2001] for directions.

Hippies problem in Promela (3)

```
active proctype Holland()
{
  bit here[N];
  hippie  h1, h2;
  do
  ::  germany_to_holland ? h1, h2;
      here[h1] = 1;
      here[h2] = 1;
      stopwatch ! max(h1,h2);
      IF all_here -> break FI;
      select_hippie(h1);
      holland_to_germany ! h1
  od
}
```

Process "`Holland`" is just the dual of "`Germany`".

```
#define max(x,y) \
  ((x>y) -> x : y)
```

```
#define all_here \
  (here[0]+here[1]+ \
   here[2]+here[3]==4)
```

```
#define select_hippie(x)  \
if                        \
:: here[0] -> x=0         \
:: here[1] -> x=1         \
:: here[2] -> x=2         \
:: here[3] -> x=3         \
fi ; here[x] = 0
```

Hippies problem in Promela (4)

```
active proctype Timer()
{
end:
  do
  ::   atomic { stopwatch ? 0 -> time=time+5 ; }
  ::   atomic { stopwatch ? 1 -> time=time+10; }
  ::   atomic { stopwatch ? 2 -> time=time+20; }
  ::   atomic { stopwatch ? 3 -> time=time+25; }
  od
}
```

Now we can use SPIN to check:
`◊(time>=120)`
`◊(time>=100)`
...

Until we find the smallest `min` for which
`◊(time>=min)`
holds.

Scheduling with SPIN (1)

- M = model of the problem in Promela
  - with (local) costs (or time) added to (some) states/transitions
  - a global variable cost is updated when a transition is taken or a state is reached.
- Goal: find schedule to an end-state with minimum cost.
  1. Verify that M is error-free.
  2. Find optimal schedule:

"eventually cost will be larger than min"

```
min = guess of (worst case, maximum) cost
do
      verify  M |= ◊ (cost ≥ min)
      if (error)  min = cost fi
while (error)
```

If there is a path to a final state for which the cost is less than min, SPIN will generate an error trail leading to this state.

Scheduling with SPIN (2)

- Idea of using (plain) model checkers for solving scheduling problems has been taken up. See for instance:

  [Brinksma & Mader - SPIN 2000]
  [Larsen et. al. – CAV 2001]
  [Ansgar Fehnker - PhD Thesis 2002]

Model Checkers are being used for serious scheduling problems!

- Original idea works, but is inefficient:
  - the (initial) complete state space already contains the most optimal solution;
  - iteratively checking ◊ (cost ≥ min) to obtain this solution is not needed, of course.

However, due to SPIN's on-the-fly model checking algorithm, for each subsequent iteration, less of the state space has to be checked: SPIN stops when it finds a state for which cost ≥ min holds.

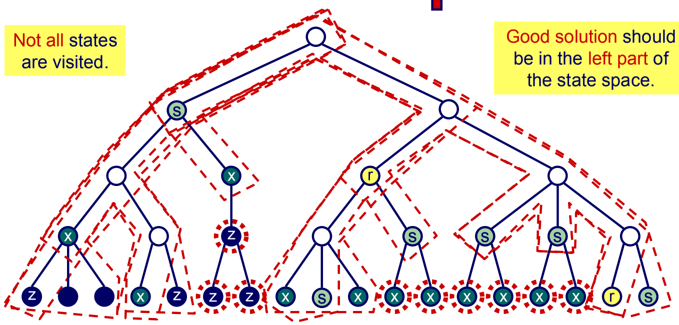
Scheduling with SPIN (3)

Example: `MAXINT` > z > x > s > r

We iteratively check (`<> cost >= min`)

Not all states are visited.

Good solution should be in the left part of the state space.



Scheduling with SPIN (4)

- If we could make the best cost "global" to all execution paths, we could update this best cost each time we find a better one.
- Sketch to find an optimal schedule in SPIN version 3.x:
  - Add a global variable `best_cost` to `pan.c` that is global for all verification runs; the variable `best_cost` will not be part of the state vector.
  - Everytime a schedule is found of which the cost is lower, the variable `best_cost` is updated and the trail leading to this schedule is saved.
  - The variable `best_cost` is initialised in a special section before the verification is started.

Drawback: the C source code of the pan verifier (or of SPIN itself) has to be modified.

SPIN 4.0 (1)

- SPIN 4.0 supports the inclusion of embedded C code into Promela models. Five new primitives:

`c_decl`	to introduce C types that can be used in the Promela model
`c_state`	to add new C variables: Global, Local or Hidden
`c_expr`	to execute a C expression and use the return value in the model
`c_code`	to add atomic C statements to the model
`c_track`	can be used to track memory, holding state information

SPIN 4.0 (2)

- The purpose of the new primitives is to provide support for automatic model extraction from C code.
  - to build your "own" FeaVer [Holzmann 2000] .
- … "The capability to embed arbitrary fragments of C code into a Promela model is very powerful and therefore easily misused" …

But we can safely use it to find the optimal solution for an optimizing problem, like the "hippies" problem.

Another feature of SPIN 4.0: **pan** now has a "guided simulation" mode. It is not longer needed to replay the simulation with **spin**.

Hippies in SPIN 4.0 (1)

Just printing the time of the schedules found.

```
proctype Holland()
{
  bit      here[N];
  hippie   h1, h2;
  do
  ::  unsafe_to_safe ? h1, h2;
      here[h1] = 1;
      here[h2] = 1;
      stopwatch ! max(h1, h2);
      IF all_here -> break FI;
      select_hippie(h1);
      safe_to_unsafe ! h1
  od ;
  c_code {
      printf("found another schedule: %d\n", now.time);
  }
}
```

"**now**" is the current state; we can access the global and local variables via this C variable.

we could also save the schedules by calling **pan**'s **putrail()**

Hippies in SPIN 4.0 (2)

Only saving the best schedule.

(1) In the declaration part of the Promela file:

```
c_state "int  best_time" "Hidden" "1000"
```

This declaration is copied verbatim to pan.c.

initial value

"Hidden" means that the variable is not stored in the state vector, but is global for the whole verification.

(2) At the end of the **Holland** proctype (i.e. in the final state of a possible schedule):

```
c_code {
    if (now.time < best_time) {
        best_time = now.time;
        printf("> best time now: %d\n", best_time);
        putrail();
        Nr_Trails--; /* only save the best trail */
    };
}
```

putrail and **Nr_Trails** are globals of **pan.c**.

Hippies in SPIN 4.0 (3)

Optimization: simple (branch &) bound.

Simple optimization:

- If after the "stopwatch has been pressed", the `time` is already greater than `best_time`, we know that this execution trace will not lead to a better trace. So, we could stop searching the state space.

- In the `Germany` proctype:

```
...
stopwatch ! h1;
if
:: c_expr { (now.time > best_time) } -> break
:: else
fi;
```

executable if the C expression is non-zero

- In the `Holland` proctype:

```
...
stopwatch ! max(h1,h2);
if
:: c_expr { (now.time > best_time) } -> break
:: else
fi;
```

Beware: we are now changing the model; invalid end-states will get introduced!

Hippies in SPIN 4.0 (4)

Optimization: (branch &) bound in never claim.

Branch & bound in never claim:

- Instead of pruning the search tree from within the Promela model, we can also limit the search of the state space via a `never` claim (i.e. combination with original idea).

- We *on-the-fly* check ◊ (now.time ≥ best_time):

```
#define higher_cost (c_expr { now.time >= best_time })

never {     /* !<> higher_cost */
accept_init:
T0_init:
    if
    :: (! (higher_cost)) -> goto T0_init
    fi;
}
```

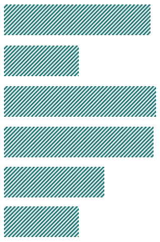
Note that the property we are checking is dynamically changed during the verification!

SPIN verifies a LTL formula using a never claim, that is automatically generated from the formula.

Hippies in SPIN 4.0 (5)

Verification results with SPIN 4.0

version	*description*	*# states*
original	deadlock checking	**2541**
original	◊ (time ≥ 60)	**1325**
hippies 1	just print all schedules found	**2659**
hippies 2	saving the best schedule	**2630**
hippies 3	branch & bound in processes	**1766**
hippies 4	branch & bound in never claim	**1330**



With SPIN 4.0, the traversal of the state space can be influenced:
- in the Promela model
- but (even more conveniently and efficiently) in the property

General procedure

- To find the optimal solution to a integer problem specified in Promela, change the model such that when a solution is found
  - the hidden `c_state` variable `best_cost` is updated
  - the path corresponding to this solution is saved

  then use SPIN to check:

```
<> higher_cost
```

where

```
#define higher_cost (c_expr {
  (now.cost >= best_cost) || \
  (will_not_be_better()) \
})
```

Branch & Bound: the C function `will_not_be_better` "looks into the future": it returns a non-zero value if given the current state, the best possible remainder will be worse than the `best_cost` so far.

Reordering the model (1)

- How to get good solutions in the left part of the search tree?
  - … by only modifying the Promela model
- SPIN's basic depth-first-search algorithm

```
procedure dfs(s: state)
    if error(s) then report error fi
    add s to Statespace
    foreach successor t of s do
        if t not in Statespace then dfs(t) fi
    od
end dfs
```

Only in the selection of the successors we can influence the DFS algorithm.

SPIN orders the list of successors as follows:
- **processes** are arranged in reverse order of creation
- within each process, all possible **executable statements** (i.e. `if` or `do`) are arranged in normal order

Reordering the model (2)

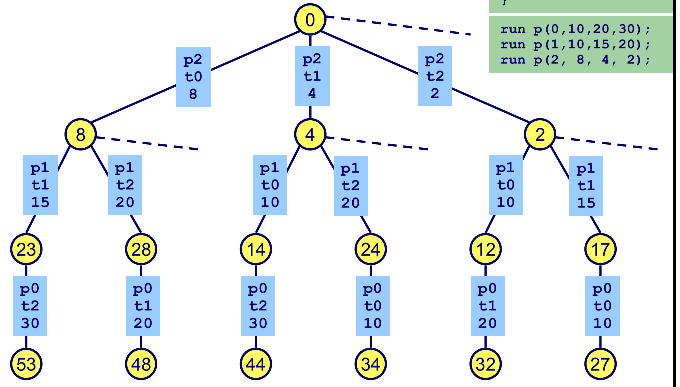
- Example:
  - 3 processors, 3 tasks, only 1 task at a time
  - each processor can only process a single task

```
bit done[3]; byte time;

#define DOTASK(i,t) \
  atomic { !done[i] -> time = time+t; done[i] = true }

proctype p(byte id; byte t0, t1, t2) {
  if
  ::  DOTASK(0, t0)
  ::  DOTASK(1, t1)
  ::  DOTASK(2, t2)
  fi
}
init { atomic { run p(0, 10, 20, 30);
                run p(1, 10, 15, 20);
                run p(2,  8,  4,  2);
}}
```

Reordering the model (3)

```
proctype p(...) {
  if
  ::  DOTASK(0, t0)
  ::  DOTASK(1, t1)
  ::  DOTASK(2, t2)
  fi
}
```

```
run p(0,10,20,30);
run p(1,10,15,20);
run p(2, 8, 4, 2);
```

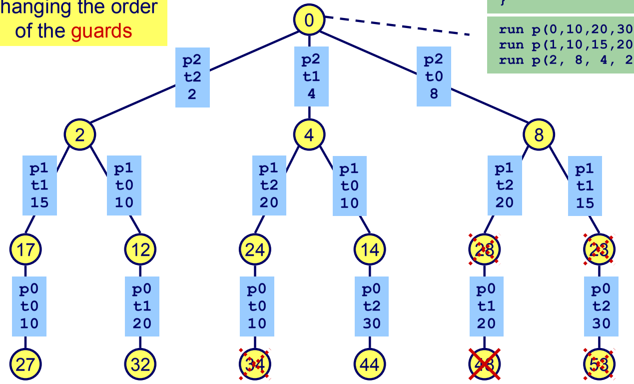


Reordering the model (4)

```
proctype p(...) {
  if
  ::  DOTASK(2, t2)
  ::  DOTASK(1, t1)
  ::  DOTASK(0, t0)
  fi
}
```

```
run p(0,10,20,30);
run p(1,10,15,20);
run p(2, 8, 4, 2);
```

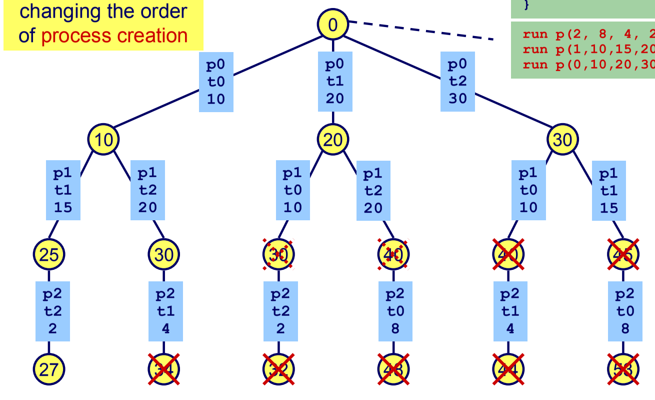
changing the order of the guards



Reordering the model (5)

```
proctype p(...) {
  if
  :: DOTASK(0, t0)
  :: DOTASK(1, t1)
  :: DOTASK(2, t2)
  fi
}
```

```
run p(2, 8, 4, 2);
run p(1,10,15,20);
run p(0,10,20,30);
```



changing the order of process creation

Summary – on optimization problems

- The model checking approach to find an optimal solution to an integer optimization problem is appealing:
  - First use the model checker to verify that the formalisation of the problem is correct.
  - Then use the model checker to obtain an (optimal) solution to the problem.
- SPIN 4.0 offers nice features to implement the Branch & Bound approach on the Promela level.
  - The Branch & Bound functionality can elegantly and efficiently be isolated in the property being checked.
  - By reordering the Promela model, we can further improve the search dramatically:
    - For certain problems to find the optimal solution, less than 5% of the states had to be visited.

Overview

Part 1

- Introduction
- Effective SPIN: the art of Promela Modelling
- Checking invariance
- Systematic Verification
- Solving optimisation problems with SPIN 4.0

Part 2

- How SPIN works:
  - Some automata theory
- Complexity issues
  - Reduction and compression
- Model extraction
  - Software model checking

How Spin Works

Basic Verification Method

1. Construct/derive an abstract model of a system.
2. Formalize its correctness properties.
3. Run the model checking algorithm.
4. Interpret the result
   - the model satisfies the property
   - the model can violate the property
   - there were insufficient resources to solve the problem (interpretation: insufficient abstraction: find a better model)
5. Revise 1 or 2 and repeat 3-5 until done…

The One-Slide Theory

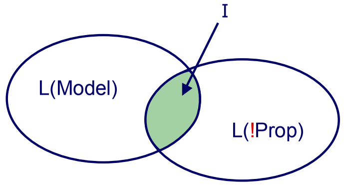
- System: L(Model)
- Requirement: L(Prop)
- Show that: L(Model) $\subseteq$ L(Prop)
- Method:

  L(Model) $\cap$ (Σ^{ω} \ L(Prop)) = $\varnothing$

- i.e.:

  L(Model) $\cap$ L(**!** Prop) = $\varnothing$

Intersection of 2 formal languages

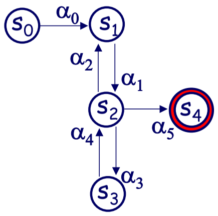


if I is empty:	the Model satisfies the Property
if I is non-empty:	the Model can violate the Property and I contains a counter-example

Finite automata

- A finite automaton is a tuple {S, s_0, L, F, T}
  - S finite set of states
  - $s_0 \in S$ initial state
  - L finite set of labels (symbols)
  - $F \subseteq S$ set of final states
  - $T \subseteq SxLxS$ transition relation

An example



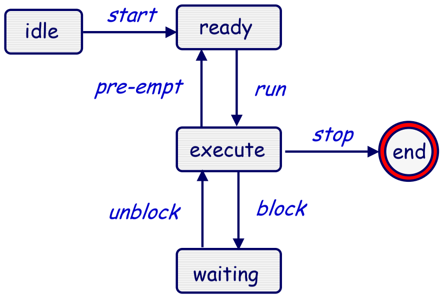
S: { s_0, s_1, s_2, s_3, s_4 }

L = { α_0, α_1, α_2, α_3, α_4, α_5 }

F = { s4 }

T = {(s_0,α_0,s_1), (s_1, α_1,s_2), ...}

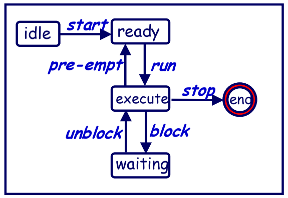
An interpretation



The definition of a run

a *run* of automaton {S, s_0, L, T, F} is an ordered set
$\sigma = \{s_0, s_1, s_2, \ldots, s_k\}$
such that
$\forall i, 0 \leq i < k : \exists \alpha, \; \alpha \in L$ and $(s_i, \alpha, s_{i+1}) \in T$.

each run corresponds to one or more *words* over L



run: { idle, ready, execute, waiting, execute, end }

word: {*start, run, block, unblock, stop*}

Standard acceptance

acceptance

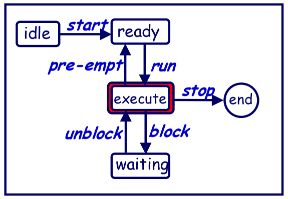
a finite run $\sigma = \{s_0, s_1, s_2, \ldots, s_k\}$
of automaton {S, s_0, L, T, F} is accepted iff
its final state $s_k \in$ **F**.

formal language

the language of automaton {S, s_0, L, T, F} is
the set of all *words* over L corresponding to
accepted runs of the automaton

Omega acceptance

an infinite run $\sigma = \{s_0, s_1, s_2, \ldots\}$
of automaton {S, s_0, L, T, F} is accepted iff
$\exists \, s_k \in$ **F**, s_k appears infinitely often in σ.



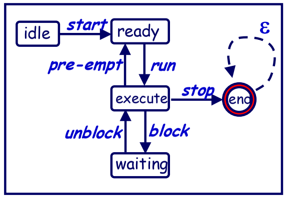
run: { idle, [ready, execute,]* }

word: {*start*, [*run, pre-empt*,]* }

language:
set of all ω-words accepted

The stutter extension rule

- a finite run can be extended into an infinite run by stuttering the final state (on a no-op ε-symbol)



run: { idle, ready, execute,
waiting, execute, [end,]* }

word: {*start, run, block,*
unblock, stop, ε* }

Logic properties

- We need a precise way to express the properties of a run
  - a little language to state desired properties of concurrent systems concisely
- Propositional linear temporal logic (LTL)
  - introduced by Amir Pnueli in late 70s
  - based on work in 'tense logics' in 50s and 60s
  - direct link with theory of ω-automata

- Example:
  - [] ((a != b) -> <> (a == b))
  - it is always the case [] that when (a != b) eventually <> we must have (a == b)
  - this defines a class of executions, rather than an instance

LTL syntax (as used in spin)

LTL formula ::=
true, false
propositional symbols p, q, r, …
(f)
unary f
f binary f

unary ::=

[]	--- always, henceforth
<>	--- eventually
X	--- next
!	--- logical negation

caution

binary ::=

U	--- strong until
&&	--- logical and
||	--- logical or
->	--- implication
<->	--- equivalence

LTL semantics

run: $\sigma = \{ s_0, s_1, s_2, s_3 \dots \}$

propositional symbols: $p, q, \dots$
$\forall i, (i \geq 0)$ and $\forall\ p,\ s_i \models p$ is defined
temporal formulae: $e, f, \dots$

→ $\sigma \models f$ iff $s_0 \models f$

with:
$s_i \models [\,]f$ iff
$\forall j, (j >= i) : s_j \models f$
$s_i \models \langle\rangle f$ iff
$\exists j, (j >= i) : s_j \models f$
$s_i \models e\ U\ f$ iff
$\exists j, (j >= i) : s_j \models f$ and
$\forall k, (i <= k < j) : s_k \models e$

Typical LTL formulae

`[]p`	always p	invariance
`<>p`	eventually p	guarentee
`p -> (<>q)`	p implies eventually q	response
`p -> (q U r)`	p implies q until r	precedence
`[]<>p`	always, eventually p	recurrence (progress)
`<>[]p`	eventually, always p	stability (non-progress)
`<>p -> <>q`	eventually p imlies eventually q	correlation

useful equivalences:

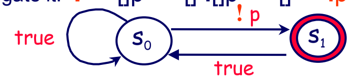
```
![]  p <=> <> !p
!<> p <=> [] !p
```

avoiding X:
next-time-free properties
are stutter-invariant

From logic to automata

- for any LTL formula f there exists a Büchi automaton that accepts precisely those ω- runs for which f is satisfied
- example: the formula `<>[]p` corresponds to the non-deterministic Büchi automaton:

- to turn a property f into a claim (the complement of f), it suffices to negate it: ! <>[]p ≡ [] ![]p ≡ []<> !p



Automata theoretic verification

- System: L (system)
- Property: L (prop)
- Show: L (system) ⊆ L (prop)
- Or that: L (system) ∩ (L^ω \ L (prop)) = ∅

no strongly connected component in the reachability graph of the intersection of claim and system should contain an accepting state

(any cycle through an accepting state would be an accepting ω-run)

Reachability by depth-first search

```
explore()
{       store = {};
        dfs(s0)
}
dfs(s)
{
        if s ∈ store
                return;
        else
                store = store ∪ {s};

        foreach successor s' of s
                dfs(s');
}
```

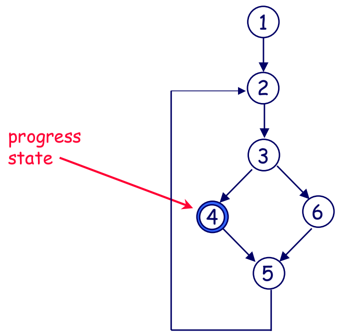
- recursively explore the state graph
- store as little data as possible
- no need to store transitions
- need not store all states
- in approximate searches, need not store states accurately

Cycle detection (1)

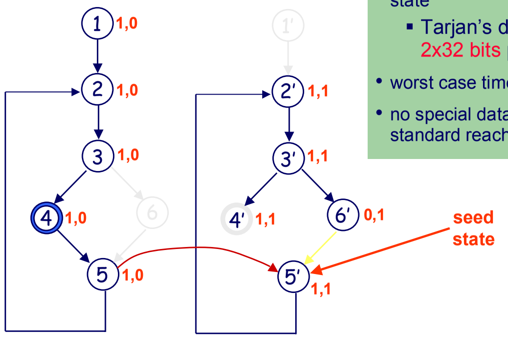
- errors (violations of an LTL property) correspond to runs with infinitely many accepting states
- in a finite graph, these correspond to reachable strongly connected components (scc's) with accepting states
- Tarjan's algorithm constructs the scc's in polynomial time -- can check each for the presence of accepting states
- but, we can do better:
  - it suffices to prove that no reachable accepting state is reachable from itself
  - same complexity, smaller constant factor

Cycle detection (2)

example: prove absence of non-progress cycles

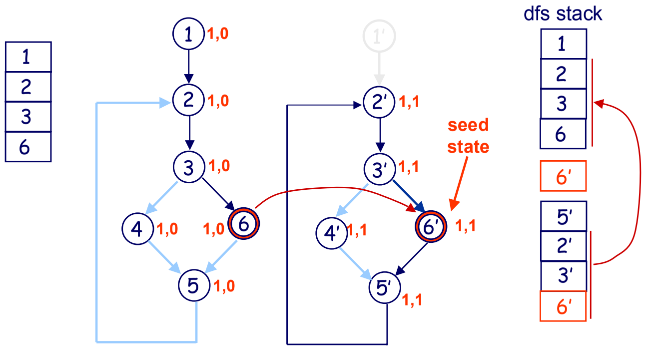


Nested depth-first search



- memory overhead: 2 bits per state
  - Tarjan's dfs requires 2x32 bits per state
- worst case time: 2x dfs
- no special data-structures - standard reachability

Detecting acceptance cycles



Cost

number of states: R

memory requirements per state: S

R x S default memory cost

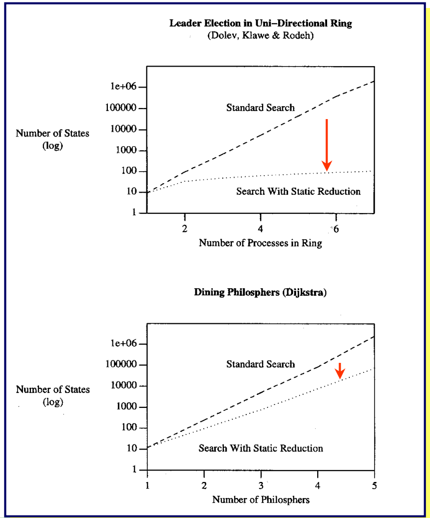
strategies for reducing R
- abstraction (model reduction)
- partial order reduction, symmetry reduction, etc.

strategies for reducing S
- lossless memory compression
- don't store states but a minimized DFA recognizer
- lossy compression (proof approximation)compute

Partial order reduction

- many runs are equivalent under given interpretation
- two transitions are independent at state s if
  - both are enabled at s
  - the execution of neither can disable the other
  - the combined effect of both transitions is independent of the relative order of execution
- strong independence
  - two transitions are strongly independent if they are independent at every state where both are enabled
- safety (a static property…)
  - a transition is safe if it is strongly independent from all other transitions in the system
  - a statement is conditionally safe for condition c if it is safe in all states where c holds

Effect of partial order reduction


Leader Election in Uni–Directional Ring
(Dolev, Klawe & Rodeh)

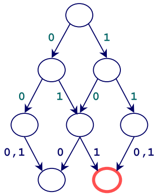
Dining Philosphers (Dijkstra)

Minimized dfa storage in spin

state descriptors can be stored as a minimized deterministic finite automaton

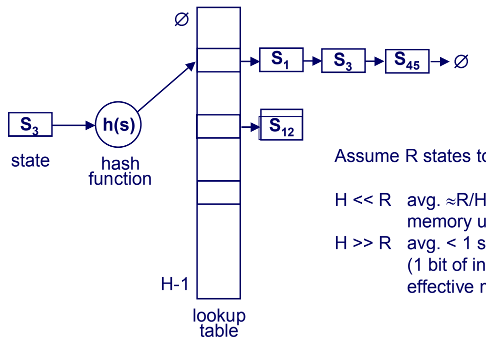
example:

states = { 011, 101, 110, 111 }



adding a new state s can be done in time O(|s|)
time/memory tradeoff:
- can reduce memory use exponentially
- more time consuming than explicit storage

State storage: hash-tables



Assume R states to be stored

H << R avg. ≈R/H states/slot
memory use R.S+overhead

H >> R avg. < 1 state/slot
(1 bit of information/slot)
effective memory use R bits

Robert Morris 1968

- assume H >> R, no need to store hash-key
- possibility of a collision becomes remote
- *“no-one to this author’s knowledge has ever implemented this idea, and if anyone has, he might well not admit it.”* [Morris, CACM1968]
- even better: use k>1 independent hash-functions
  - “store” each state k times
  - hash-collision now requires k matches
  - spin uses 2 out of 32 possible CRC polynomials

Burton Bloom 1970

- k independent hash-functions
- initially the hash-table has all zero bits.
- after r states have been stored, the probability of a specific bit being zero is:

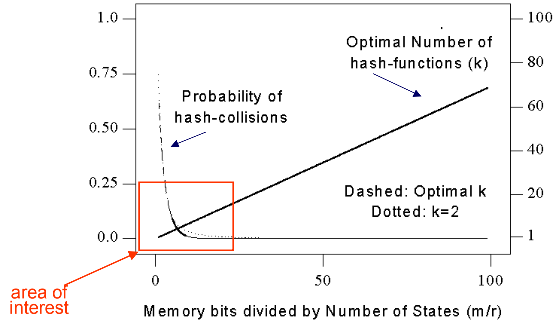
$$\left(1 - \frac{1}{m}\right)^{k.r}$$

probability of a hash-collision on (r+1)th entry:

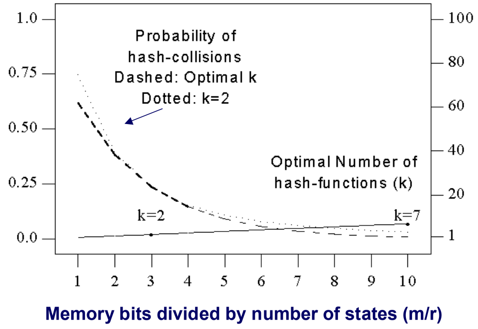
$$\left(1 - \left(1 - \frac{1}{m}\right)^{k.r}\right)^{k} = \left(1 - e^{-k.r/m}\right)^{k}$$

rhs is minimized for k = ln 2 x m/r

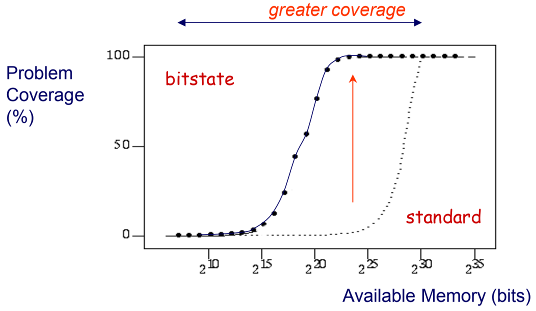
The optimal number of hash-functions



Close-up view



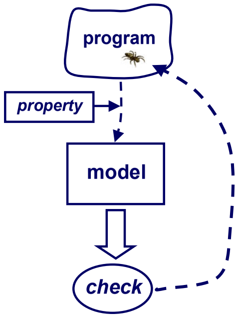
The effect of bitstate hashing



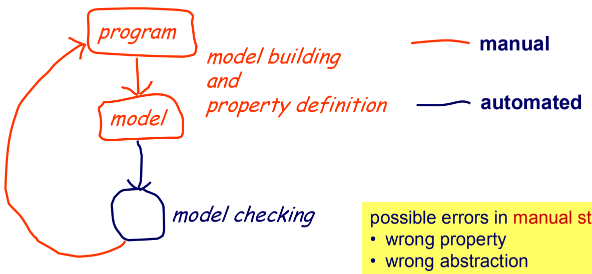
(Data: a Commercial Data Transfer Protocol)

Software verification

- most properties of interest are in *general* undecidable
- map the problem into a different domain: apply abstraction
- the best level of abstraction depends on the property to be proven
- to what extent can we automate program abstraction?



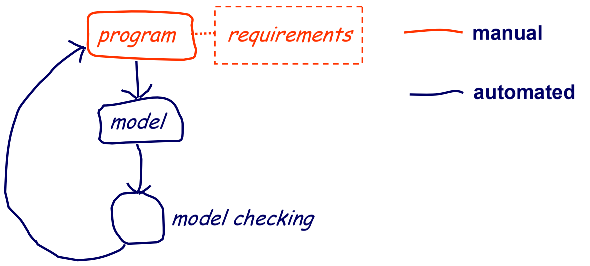
Classic model checking



possible errors in manual steps:
- wrong property
- wrong abstraction
- wrong interpretation of trail
- (or true error in program…)

possible errors in automated steps:
- model checker error

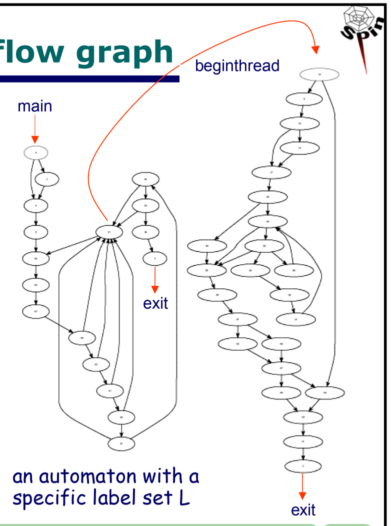
Model extraction



possible errors in manual steps:
- error in program or
- error in requirements

Program control flow graph

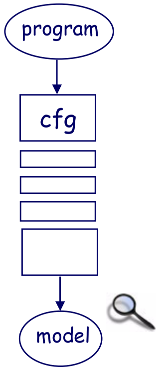
```
int
main(int argc, char *argv[])
{       char buf[512];
        int ns;

        if (!socket setup((unsigned short) TB SCHED))
                exit(1);    /* err msg already given */

        if (argc > 1)
                preload();

        printf("tb_sched: ready...\n");
        fflush(stdout);
        for (;;)
        {       sanity();
                fflush(stdout);
                ns = wait_for_request(buf, sizeof(buf));
                u cnt++;    /* update count - ok if it wraps */

                if (strncmp(buf, "offer ", strlen("offer ")) == 0)
                {       char *n = &buf[strlen("offer ")];
                        char *g = strrchr(buf, ' ');
                        if (g) { *g = '\0'; g++; }
                        add worker(fromaddress, n, g);
                }
                else if (strncmp(buf, "withdraw", strlen("withdraw")) == 0)
                        del worker(fromaddress);
                else if (strncmp(buf, "jobdone", strlen("jobdone")) == 0)
                        job done(fromaddress);  /* from prepjob */
                else if (strncmp(buf, "done ", strlen("done ")) == 0)
                        task done(fromaddress, &buf[strlen("done ")]);
                else if (strncmp(buf, "trail ", strlen("trail ")) == 0)
                        do_wrap(fromaddress, &buf[strlen("trail ")]);
                else if (strncmp(buf, "status", strlen("status")) == 0)
                        status();
                else if (strncmp(buf, "restrict", strlen("restrict")) == 0)
                        restricted = 1;
                else if (strncmp(buf, "release", strlen("release")) == 0)
                        restricted = 0;
                else if (strncmp(buf, "exit", strlen("exit")) == 0)
                        break;
                else  /* default */
                        add task(buf);

                 beginthread(assignments, 0, NULL);

                closesocket((SOCKET) (ns-1));
        }
        printf("tb_sched: exits...\n");
        exit(0);
}
```



an automaton with a specific label set L

Model extraction

- parsing
- interpretation
  - slicing
  - abstraction
  - generalization
  - restriction
- simplification
- model generation



Syntactic conversion

C

```
int c, nl, nw, nc;

main(void)
{   int inword = 0;

    while(c = getchar())
    {
        nc++;
        if (c == '\n')
            nl++;
        if (c == ' ' || c == '\n')
            inword = 0
        else if (inword == 0)
        {
            nw++;
            inword = 1;
        }
    }
    printf("%d\t%d\t%d\n", nl, nw, nc)
}
```

Promela

```
int c, nl, nw, nc;

init {
    bool inword = false;

    do
    :: STDIN?c -> nc++;
       if
       :: c == '\n' -> nl++
       :: else
       fi;
       if
       :: c == ' ' || c == '\n' ->
          inword = false
       :: else ->
          if
          :: inword == false ->
               nw++; inword = true
          :: else /* do nothing */
          fi
       fi
    od;
    printf("%d\t%d\t%d\n", nl, nw, nc)
}
```

Slicing

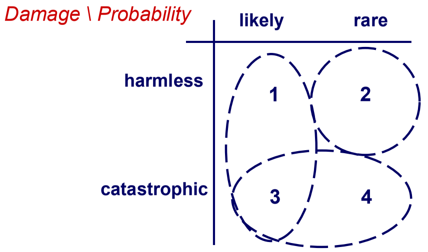
```
int c, nw, nl, nc;

init {
    bool inword = false;

    do
    :: STDIN?c -> nc++;
        if
        :: c == '\n' -> nl++
        :: else
        fi;
        if
        :: c == ' ' || c == '\n' ->
            inword = false
        :: else ->
            if
            :: inword == false ->
               nw++; inword = true
            :: else /* do nothing */
            fi
        fi
    od;
    printf("%d\t%d\t%d\n", nl, nw, nc)
}
```

property: `[] (nl ≥ nc)`
slice criteria: `{nl, nc}`

- 🟥 data dependent
- 🟩 control dependent
- ☐ independent

related issues:
boundedness
assumptions about environment
n.b.: property is not satisfied
(nc and nl can wrap around max)

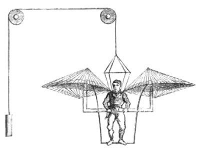
Added value of logic verification



1+3 -- covered by standard testing
3+4 -- covered by logic verification
2 -- covered but not important

Simplicity

> "Seek simplicity and distrust it."
> Alfred North Whitehead (1861-1947)



Announcement

Forthcoming (August 2003)

Gerard J. Holzmann
The Spin Model Checker
Primer and Reference Manual

Addison Wesley
ISBN 0-32122-862-6, approx. 600 pages.