Advanced SPIN Tutorial

Theo Ruys
University of Twente
The Netherlands
email: ruys@cs.utwente.nl
http://www.cs.utwente.nl/~ruys/

Gerard Holzmann
NASA/JPL Laboratory for Reliable Software
Pasadena, CA
email: gholzmann@acm.org
http://spinroot.com/gerard

SPIN 2004 Tutorial Barcelona, Spain, April 2004

Common Design Flaws

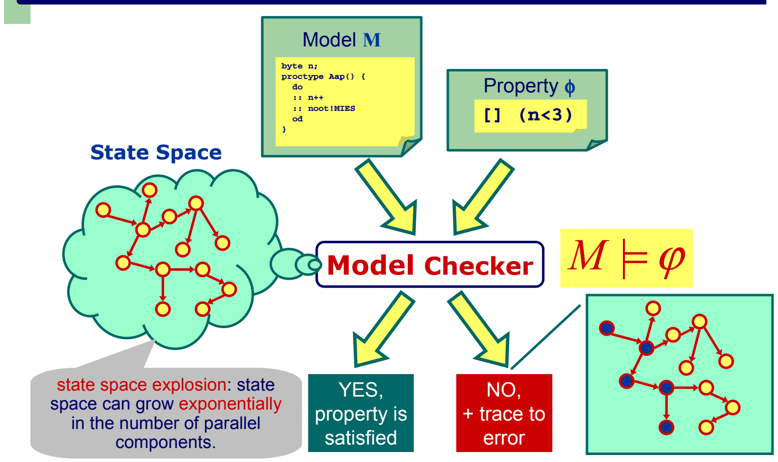
- Deadlock
- Livelock, starvation
- Underspecification
  - unexpected reception of messages
- Overspecification
  - dead code
- Violations of constraints
  - buffer overruns
  - array bounds violations
- Assumptions about speed
  - logical correctness vs. real-time performance

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 often overlooked...

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

Model Checking

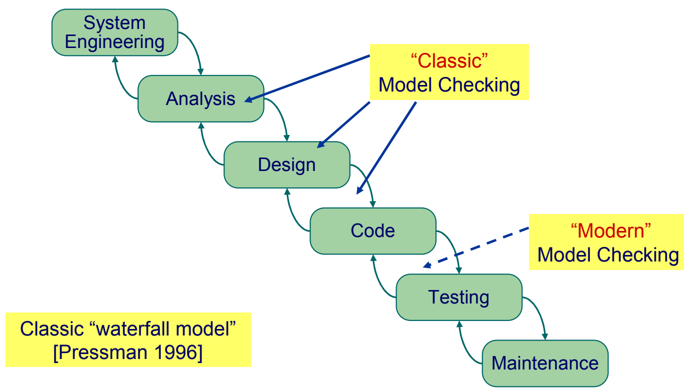


Verification vs. Debugging

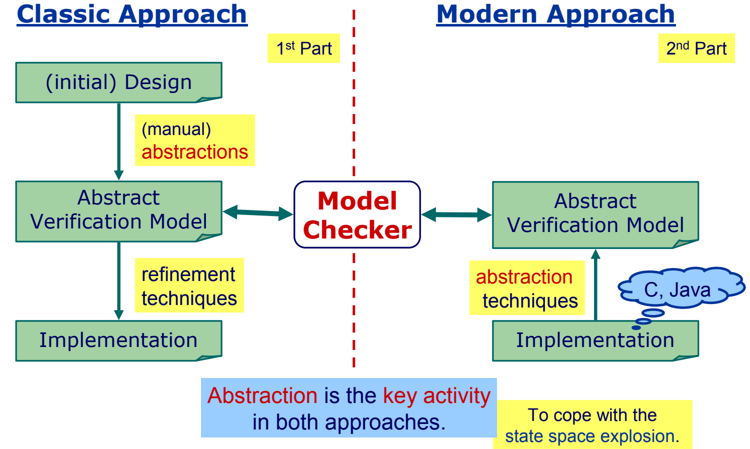
- Two approaches with respect to the application of model checkers.
  - verification approach: tries to ascertain the correctness of a detailed model M of the system under validation.
  - debugging approach: tries to find errors in a model M.
- Model checking is often most effective as a design debugging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

Automatic verification is *not* just good for proving correctness. It also excels at finding bugs very early in the design of a new system.

System Development



Classic vs Modern Approach



Overview

Tutorial - Part 1

- Introduction
- Effective SPIN: the art of Promela Modelling
- Lossy channels
- Dealing with time
- Checking Invariance
- Systematic Verification
- Solving optimisation problems with SPIN 4.x

Tutorial - 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 (first Spin book)
2.0	Jan 1995	partial order reduction
3.0	Apr 1997	minimised automaton representation
4.0	Jan 2003	embedded C code; BFS; data abstraction

- Some success factors of SPIN
  - "push the button" verification style (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

1983 Unix
1986 TeX
1997 Tcl/Tk
2001 SPIN
2002 Java

Documentation on SPIN

- SPIN's home 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 SPIN Workshops
- Gerard's website with many papers on SPIN:
  http://spinroot.com/gerard/abs.html
- SPIN version 1.0 is described in [Holzmann 1991]

available in PDF from spinroot.com

New book: Gerard J. Holzmann
The Spin Model Checker
Primer and Reference Manual
Addison Wesley, 2004
ISBN 0-32122-862-6, 608 pages.
describes SPIN up to version 4.0.

Promela Model

- A Promela model consist of:
  - type declarations — **mtype**, constants, typedefs (records)
  - channel declarations — `chan ch = [cap] of {type, ...}`
    asynchronous: *cap* > 0
    rendez-vous: *cap* == 0
  - global variable declarations
    - simple vars
    - structured vars
    - vars can be accessed by all processes
  - process declarations — behavior of the processes: local variables + statements
  - [**init** process] — initialises variables and starts processes

Promela constructs

most important Promela constructs
are either executable or blocked

six basic statements	*assignment*	always executable
	expression	executable if non-zero (i.e., *true*)
	send (**ch!**)	executable if channel **ch** is not full
	receive (**ch?**)	executable if channel **ch** is not empty
	assert(*<expr>*)	always executable
	printf	always executable
expression statements	**skip**	always executable (equivalent to 1 or *true*)
	timeout	variable, *true* if no other statement is executable
compound statements	**if**	executable if at least one guard is executable
	do	executable if at least one guard is executable
	atomic { ... }	executable if first statement is executable
	d_step { ... }	executable if first statement is executable
control-flow specifiers	**goto**	jump to label
	break	exit **do**-statement

specifiers like ";" and "->", also only specify control-flow

Promela Example

```
mtype = {REQ,ACK};          -- message types (constants)
typedef Msg {               -- "record" declaration
  byte  a[2];
  mtype tp;
} ;
chan toR = [1] of {Msg};    -- channel declaration
bool flag;                  -- global variable

proctype Sender() {
  Msg m;                    -- local variable
  ...
  m.a[0]=2; m.a[1]=7; m.tp = REQ;
  toR ! m;                  -- sending a message
}

proctype Receiver(byte n) {
  Msg m;
  ...
  toR ? m;                  -- receving a message
}
                            -- creates processes
init {
  run Sender();
  run Receiver(2);
}
```

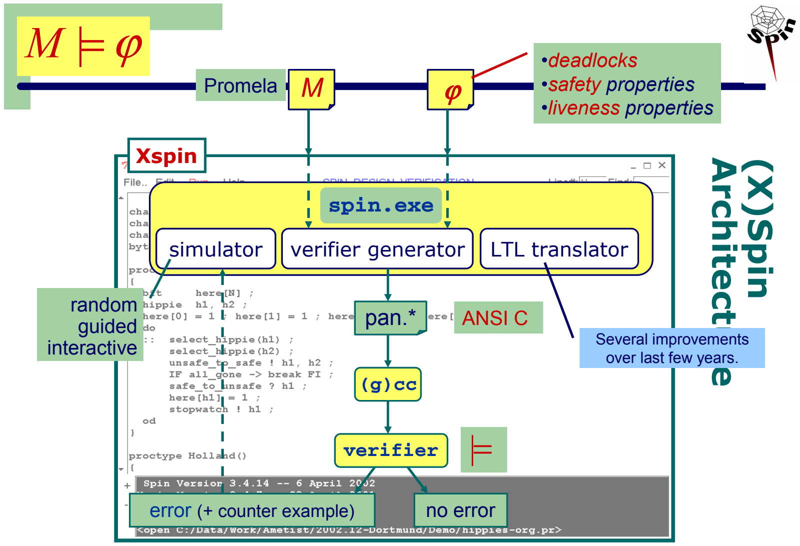
A Promela model corresponds with a (usually very large, but) finite transition system, so
- no unbounded data
- no unbounded channels
- no unbounded processes
- no unbounded process creation

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 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 variants like HashCompact

$M \models \varphi$ (X)Spin Architecture



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
```

Visit all processes and collect all executable actions .

interactive simulation: act is chosen by the user

act is executed and the system enters a new state

$s \xrightarrow{act} t$

Verification Algorithm (1)

$s \xrightarrow{act} t$

- 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 hash table

state matching

Only works for safety properties.

the old states s are kept on the dfs-stack, which corresponds with a complete execution path

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

Process automaton (1)

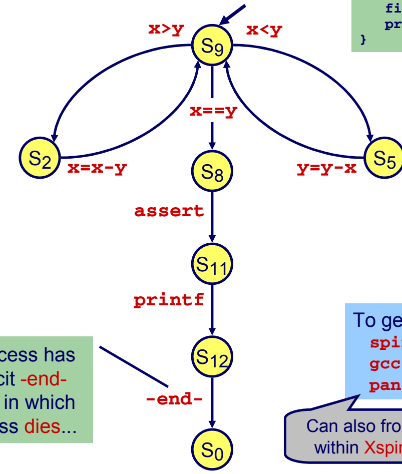
- Every promela **proctype** defines a finite state automaton, (S, s0, L, T, F), where
  - S is a set of states
  - s0 is the initial state, s0 ∈ S
  - L is a finite set of labels
  - T is a set of transitions, T ⊆ S × L × S
  - F is a set of final states, F ⊆ S

A label *l* ∈ L is one of the six basic statements:
- assignment
- **assert**
- **printf**
- send (!)
- receive (?)
- expression

other Promela statements serve to specify possible flow of control, i.e. the transition relation T.

Process automaton (2)

```
proctype gcd(int x, y) {
L: if
   :: (x > y)  -> x = x-y; goto L
   :: (x < y)  -> y = y-x; goto L
   :: (x == y) -> assert(x == y);
   fi;
   printf("gcd = %d\n", x)
}
```



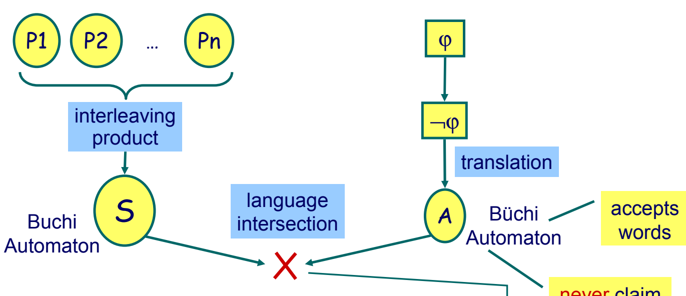
Each process has an implicit -end- transition in which the process dies...

To generate automaton:
spin -a -o3 gcd.pr
gcc -o pan pan.c
pan -d

Can also from within Xspin

-o3 is to disable statement merging

Verification Algorithm (2)



The intersection X should be empty. If non-empty, Spin searches for a reachable accepting state in the intersection that is reachable from itself. This is implemented by a nested DFS procedure. See [Holzmann 1996 et. al. – DFS] 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:
  - all global variables
  - contents of all message channels
  - for each process in the system:
    - all local variables
    - the process counter of the process
- It is important to minimize 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)

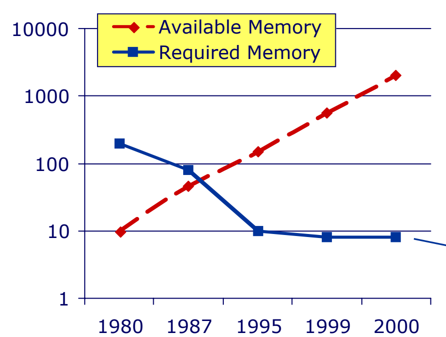
    instead of storing each state explicitly, only one bit of memory is used to store a reachable state
  - hash compaction (approximate)
  - state vector compression ("zipping the individual states")
  - minimized automaton encoding of states (much like a BDD)
  - dataflow analysis: dead variable analysis, statement merging
  - slicing algorithm ("give hints of what can be thrown away")

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

Moore's Law & Advanced Algorithms

[Holzmann 2000 M'dorf]

- Verification results of `Tpc` (The phone company)



7 days

7 secs

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

Experimentation:
- several optimising options make it difficult to predict the behaviour of a new verification run;
- there are many different ways to model a particular aspect
- use series of controlled verification runs with different settings to conclude which modelling solution works 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 optimisation problems

Added this week

- 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);
}
```


Lossy channels

- It's already difficult to design and implement systems for an ideal world in which no mistakes are made.
  - Unfortunately, users and environment are not perfect. Still the system has to be error-proof.
- Even if we restrict ourselves to a (lower level) protocol, which defines a means to transmit messages between processes, several types of errors can be introduced:
  - messages can get lost
    - These types of errors are the most common, caused by so-called "lossy channels".
  - messages can be duplicated
  - messages can be inserted
- How can we implement lossy channels in Promela?

Alternating Bit Protocol

perfect channels

```
#define MAX 4;

mtype {MSG, ACK};

chan toR = [1] of {mtype, byte, bit};
chan toS = [1] of {mtype, bit};

active proctype Sender()
{
  byte data;
  bit  sendb, recvb;

  sendb = 0;
  data  = 0;
  do
  :: toR ! MSG(data,sendb) ->
     toS ? ACK(recvb);
     if
     :: recvb == sendb ->
        sendb = 1-sendb;
        data  = (data+1)%MAX;
     :: else /* resend old data */
     fi
  od
}
```

Consequently, the else-branches will never be taken.

```
active proctype Receiver()
{
  byte data, exp_data;
  bit  ab, exp_ab;

  exp_ab   = 0;
  exp_data = 0;

  do
  :: toR ? MSG(data,ab) ->
     if
     :: (ab == exp_ab) ->
        assert(data == exp_data);

        exp_ab    = 1-exp_ab;
        exp_data  = (exp_data+1)%MAX;
     :: else
     fi;
     toS ! ACK(ab)
  od
}
```


Lossy channels - *Daemon*

Introduce Daemon process that steals messages from the channels **toS** and **toR.**

- Let the **Sender** recover from a message not received.
- Notes:
  - + original model does not have to be modified
  - – extra process: extra space in state vector
  - – receive channels may not longer be exclusive to processes

SPIN's slicing algorithm reports:
spin: proctype Daemon defines a sink process to reduce complexity, consider merging the code of each sink process into the code of its source

```
active proctype Sender()
{
  ...
  do
  :: toR ! MSG(data,sendb) ->
     if
     :: toS ? ACK(recvb);
        if
        :: recvb == sendb ->
           sendb = 1-sendb;
           data  = (data+1)%MAX;
        :: else /* resend data */
        fi
     :: timeout /* message lost */
     fi
  od
}

active proctype Daemon()
{
  do
  :: toR ? _, _, _
  :: toS ? _, _
  od
}
```


Lossy channels - *Sending process loses msgs*

Follow SPIN's advice: let each sending process that sends a message also lose the message (i.e. not sending the message).

- In the ABP example both the **Sender** and **Receiver** are 'sending processes'.
- Notes:
  - – original model must be modified
  - + no extra process needed
  - + channels may still be exclusive to processes

This means that we can use Promela's **xs** and **xr** declarations to help SPIN's partial order reduction algorithm.

```
active proctype Sender()
{
  ...
  do
  :: if
     :: toR ! MSG(data,sendb)
     :: true
     fi;

     if
     :: toS ? ACK(recvb);
        if
        :: recvb == sendb ->
           sendb = 1-sendb;
           data  = (data+1)%MAX;
        :: else /* resend data */
        fi
     :: timeout /* message lost */
     fi
  od
}
```

Each send operation in the model will be enclosed in an **if**-clause with an always enabled second guard (i.e. **true**).

Lossy channels - *Receiving proc loses msgs*

Instead of not sending a message we can also have each receiving process lose messages by ignoring messages received (i.e. not doing anything with the message).

- In the ABP example both the **Sender** and **Receiver** are 'receiving processes'.
- Notes:
  - – original model must be modified
  - – message is actually send; extra states will get introduced
  - + no extra process needed
  - + channels may still be exclusive to processes

again we can use **xs** and **xr** declarations

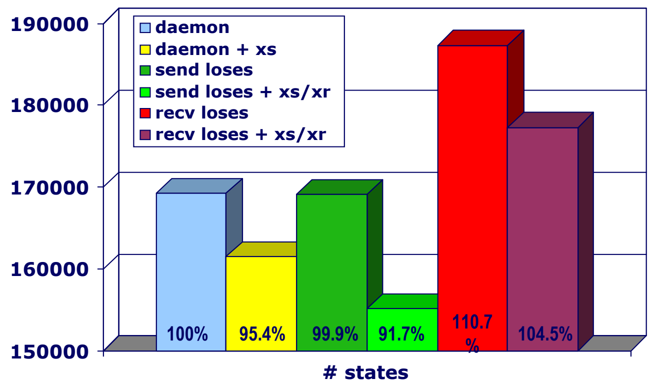
```
active proctype Sender()
{
  ...
  do
  :: toR ! MSG(data,sendb);
     if
     :: toS ? ACK(recvb);
        if
        :: recvb == sendb ->
           sendb = 1-sendb;
           data  = (data+1)%MAX;
        :: else /* resend data */
        fi
     :: toS ? _,_  /* lose message */
     :: timeout    /* message lost */
     fi
  od
}
```

Do not add a **skip**-alternative (as in the 'sending loses msg' case), as this models an premature timeout.

Lossy channels - *Experiments with BRP*

PowerBook G4 1Ghz 768Mb
Mac OS X 10.3.2 SPIN 4.1.2

state vector daemon: 92
state vector others: 88



Lossy channels - *Conclusions*

- Advice
  - Always try to use xs/xr declarations in your model.
  - To model lossy channels, simply use a “daemon process” in your preliminary models.
  - When running out of memory, implement the ‘sending process loses message’ scheme.
    - Be careful: this is not always as easy as simply adding some skips. Especially if the daemon process does something more than just stealing messages.
  - Do not use the ‘receiving process loses message’ scheme.

Timeouts (1)

- Promela is optimised for logic verification: it does not have real-time features.
  - In Promela we only specify functional behaviour.
  - But, most protocols use timers or a timeout mechanism to resend messages or acknowledgements.
- `timeout`
  - special variable in Promela
  - value of `timeout` will only be set to true in a state when there is no other statement in the system which is executable.
    - so, `timeout` models a global timeout.
  - `timeout` provides an escape from deadlock states.
  - beware of statements that are always executable…
- `else`
  - is also a special variable (!) which can be used in `if`/`do` statements
  - value is only true when all other guards of the `if`/`do` statement in which it appears are non-executable
  - matches intuition for standard if-then-else style constructs

Timeouts (2)

- General scheme to recover from message loss:

```
proctype Sender() {
  ...
  do
  :: toR ! MSG(data,sendb) ->
       if
       :: toS ? ACK(recvb) -> ... /* normal case      */
       :: timeout ->          ... /* message got lost */
       fi
  od
}
```

Premature timeouts can be modelled by replacing the `timeout` by a `skip` (which is always executable).

One might want to limit the number of premature timeouts, though… [Ruys & Langerak 1997].

Timeouts (3)

```
do
:: toR ! MSG(data,sendb) ->
   if
   :: toS ? ACK(recvb) -> /* normal */
   :: timerexpired ->     /* error  */
   fi
od
```

- Simple scheme to limit the number of premature timeouts:

```
#define timerexpired                                  \
  premature_timeout || timeout

#define set_premature_timeout                         \
  if                                                  \
  :: nr_prem_timeouts <= MAX_PREM_TIMEOUTS ->         \
     if                                               \
     ::  premature_timeout=true ;                     \
         nr_prem_timeouts++                           \
     ::  premature_timeout=false                      \
     fi                                               \
  :: else -> premature_timeout=false                  \
  fi
```

This macro should be called each time a premature timeout has occurred, i.e., in the `/* error */` part above.

Timeouts (4)

- Modelling timeout using special timeout channels.
  - E.g., after losing a message, the **Daemon** process also sends a timeout message to the waiting process.

```
chan timeoutCh = [0] of {bit};

active proctype Daemon()
{
  do
  :: atomic {toR ? _, _, _ -> timeoutCh!1 }
  :: atomic {toS ? _, _    -> timeoutCh!1 }
  od
}
```

```
:: toR ! MSG(data,sendb) ->
   if
   :: toS ? ACK(recvb) -> ... /* normal case      */
   :: timeoutCh?1      -> ... /* message got lost */
   fi
```


Simulating time (1)

- Instead of using an abstraction of timeouts, it is also possible to simulate (discrete) time.

```
byte time;
proctype Tick() {
  do
  :: timeout -> (time = time+1)%MAXTIME;
  od
}
```

If none of the other processes can proceed, it is time to increase the **time**.

Note: by using this **timeout**, you cannot longer search for deadlocks.

- Other processes can now wait for time to pass:

```
byte stamp;
...
do
:: atomic {toR ! MSG(data,sendb) -> stamp=time };
   if
   :: toS ? ACK(recvb) ->                        /* good */
   :: time >= (stamp + SENDER_TIMEOUT)%MAXTIME; -> /* bad */
   fi
od
```

but, beware of effects of module MAXTIME counting!

Simulating time (2)

- Using this approach to simulate time does work, but has a few disadvantages.
  - All actions that take time have to synchronise on **time**.
    - The passing of time acts like a scheduler for the model, which makes the model harder to understand.
  - Importantly, it is an expensive way of dealing with time.
    - Due to the process **Tick** the number of states can blow up.
- Variants and enhancements of the approach have been proved to work (for small cases).
  - See for instance [Bosnacki & Dams, FORTE/PSTV 1998].

Simulating time (3)

- Variable time advance
  - Well-known technique from operations research.
  - Simulated time goes forward to the next moment in time at which some event triggers a state transition, and all intervening time is skipped.

Works efficiently for scheduling problems (e.g. [Brinksma & Mader 2000], [Ruys 2003]).

This becomes especially attractive when the differences between these moments in time are large.

```
byte time;
```

global time (like with Tick)

```
byte next_time[N]
```

each process i can set its timer next_time[i] to signal that it is waiting, otherwise it is zero.

```
#define PROCESS_WAITING \
(next_time[0] + ...+ next_time[N-1]) > 0
```

this expression is non-zero if at least on process i is waiting for time to reach next_time[i].

Simulating time (4)

```
proctype TimeAdvance() {
  byte i, earliest;
  do
  :: d_step {
       PROCESS_WAITING ->
       i=0; earliest=MAX_TIME;
       do
       :: i<N -> if
                 :: next_time[i] > 0 &&
                    next_time[i] < earliest ->
                      earliest = next_time[i]
                 :: else
                 fi;
                 i++
       :: else -> break
       od;
       time=earliest; i=0; earliest=0;
     }
  od
}
```

As in the original Tick process, it is sometimes needed to guard the 'passing of time' by a timeout statement.

Note, if an action within a process *takes time*, the variable time should be updated within that particular process (beware of concurrent actions).

Simulating time (5)

- Two successful implementations of time into SPIN.
  - RT-SPIN [Tripakis & Courcoubetis 1996]
    - real-time
  - DT-SPIN [Bosnacki & Dams 1998]
    - deterministic time
    - uses similar approach like the process Tick, but has changed the partial order algorithm in SPIN to take advantage of special characteristics of the process Tick.
- When serious about verifying timing constraints, one should use a dedicated real-time model checker like UPPAAL.
  - Use them both:
    - SPIN for the functional correctness of the model (abstracting from time)
    - UPPAAL for checking the timing constraints

A disadvantage of these approaches is that they are not available for the 'latest' versions of SPIN.

Invariance

[]P

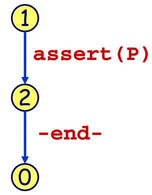
- []P where P is a state property
  - safety property
  - invariance ≡ global universality or global absence [Dwyer et. al. 1999]:
    - 25% of the properties that are being checked with model checkers are invariance properties
    - BTW, 48% of the properties are response properties
  - examples:
    - `[] !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

- 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 -> ...}`

- 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.

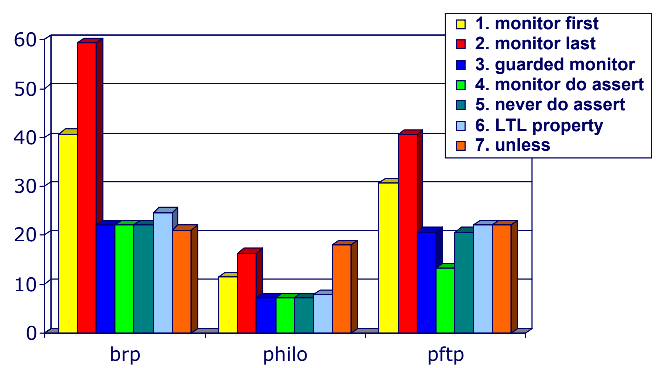
Note: disabling partial reduction (**`-DNOREDUCE`**) may have severe negative consequences on the effectiveness of the verification run.

Invariance experiments (1)

PII 300Mhz
128 Mb
SPIN 3.3.10
Linux 2.2.12

-DNOREDUCE NO partial order reduction

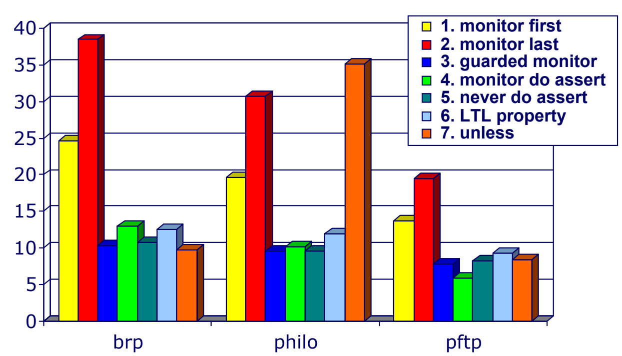
memory (Mb)



Invariance experiments (2)

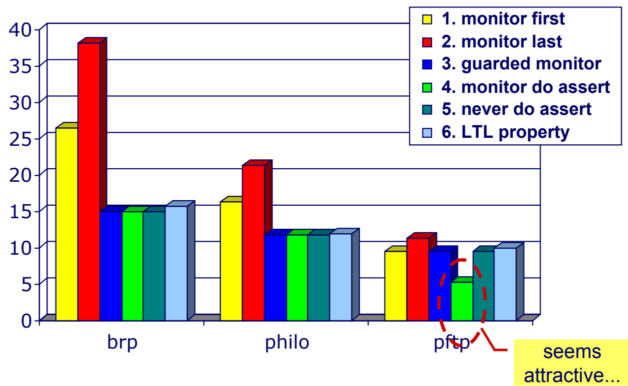
-DNOREDUCE NO partial order reduction

time (sec)



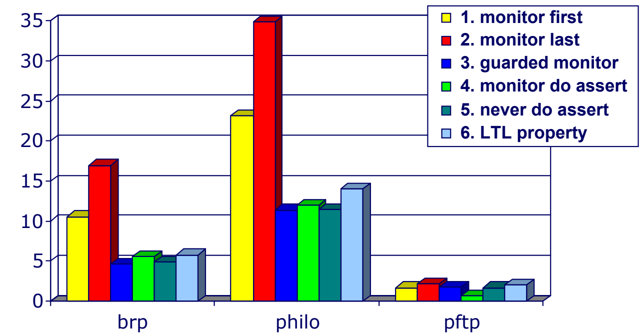
Invariance experiments (3)

default settings
memory (Mb)



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: $\mathbf{M}_i$
    - revisions: $\mathbf{M}_{i,j}$
  - 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:
  - Re-verification 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
```

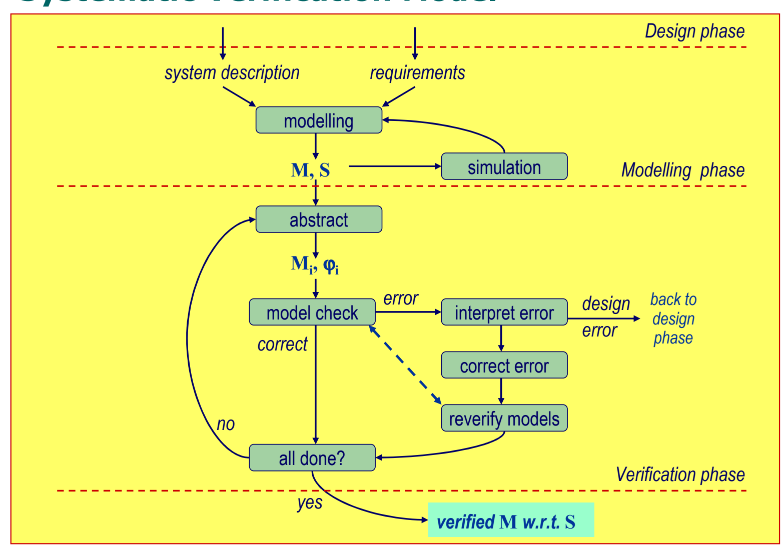
longest execution path

property was satisfied

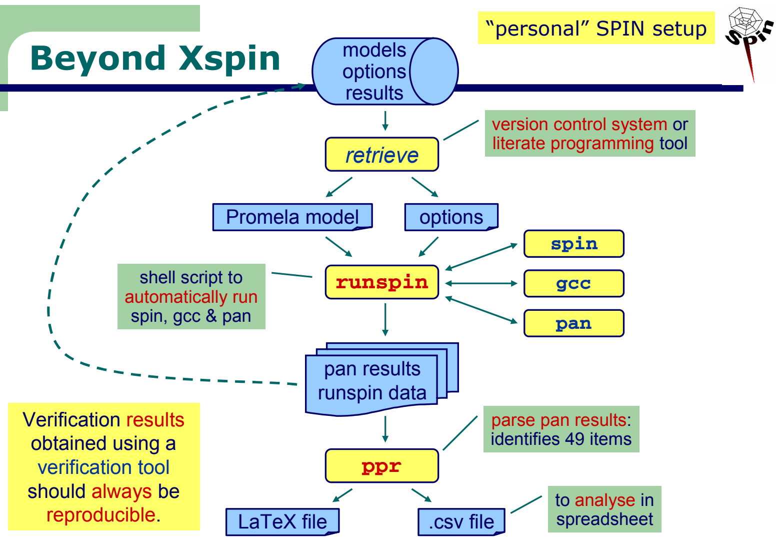
total number of states (i.e. the state space)

total amount of memory used for this verification

Systematic Verification Model



Beyond Xspin



Optimisation problems with SPIN

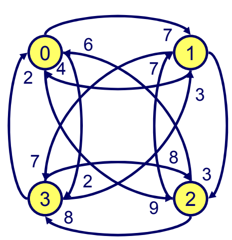
- Introduction: Traveling Salesman Problem
- SPIN 4.x - new features
- Branch & Bound with SPIN 4.x
- Reordering the Promela model

Traveling Salesman Problem

- Traveling Salesman Problem (TSP)
  - n cities
  - cost c_{ij} between city i and j
  - non-Euclidean: $c_{ij} \neq c_{ji}$
  - TSP: connect the cities with the shortest closed tour, passing each exactly once.

tour with lowest cost

Example: n=4



	0	1	2	3
0	-	7	9	2
1	4	-	3	7
2	6	7	-	8
3	2	3	8	-

cost matrix

Promela model of TSP

	0	1	2	3
0	-	7	9	2
1	4	-	3	7
2	6	7	-	8
3	2	3	8	-

Promela model:

- single process TSP
- N places/labels
- **`bit vv[N]`** (to mark visited cities)
- variable **`cost`** to hold the total cost so far
- from each label we jump non-deterministically to cities not visited yet
- when a choice is made, **`cost`** is updated

Note: it is better to use bitvectors than `bit[]` (see [Ruys 2000]).

```
local bit vv[4];
local int cost;

active proctype TSP()
{
P0: atomic {
    if
    :: !vv[1] -> cost = cost+7; goto P1
    :: !vv[2] -> cost = cost+9; goto P2
    :: !vv[3] -> cost = cost+2; goto P3
    fi
}
P1: atomic {
    vv[1] = true;
    if
    :: !vv[2] -> cost = cost+3; goto P2
    :: !vv[3] -> cost = cost+7; goto P3
    :: else   -> cost = cost+4; goto end
    fi
}
...
end:
    skip
}
```

global so we can access them in the never claim

here all places have been visited

Finding the shortest tour

- We now use the Promela model to let SPIN find the shortest tour through the cities.
- We let SPIN verify iteratively:
  - `◊(cost >= 1000)`
    not satisfied: counter example with cost = 20
  - `◊(cost >= 20)`
    not satisfied: counter example with cost = 14
  - `◊(cost >= 14)`
    satisfied
    so the minimum cost = 14

Optimisation problems 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.

Optimisation problems with SPIN (2)

- Idea of using (plain) model checkers for solving scheduling problems has been taken up. See for instance (among many others):
  [Ruys & Brinksma - TACAS 1998]
  [Brinksma & Mader - SPIN 2000]
  [Larsen et. al. – CAV 2001]
  [Ansgar Fehnker - PhD Thesis 2002]

Model Checkers are being used for serious optimisation 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.

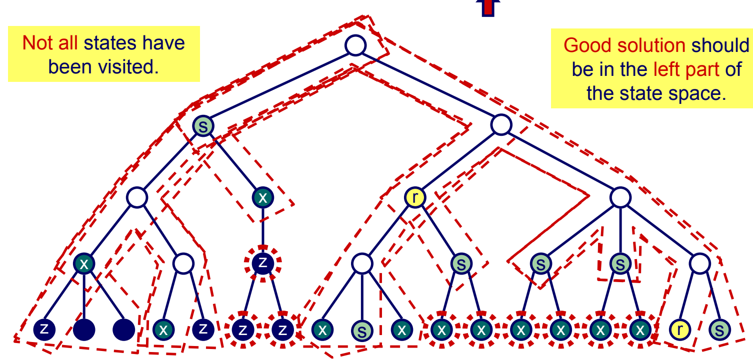
Optimisation problems with SPIN (3)

Example: MAXINT > z > x > s > r

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

Not all states have been visited.

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



Optimisation problems 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 solution 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 solution is found of which the cost is lower, the variable `best_cost` is updated and the trail leading to this solution 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.x (1)

- SPIN 4.x 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.x (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 powerful and therefore easily misused" … [Holzmann 2004, p. 501].

But we can safely use it to find the optimal solution for an optimization problem, like the TSP.

Another feature of SPIN 4.x: `pan` now has a "guided simulation" mode. It is not longer needed to replay the simulation with `spin`.

TSP in SPIN 4.x (1)

Just printing the cost of the solutions found.

```
local bit vv[4];
local int cost;

active proctype TSP()
{
...
P1: atomic {
    vv[1] = true;
    if
    :: !vv[2] -> cost = cost+3; goto P2
    :: !vv[3] -> cost = cost+7; goto P3
    :: else   -> cost = cost+4; goto end
    fi
}
...
end:
    c_code {
        printf("found another solution: %d\n", now.cost);
    }
}
```

"`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()`

TSP in SPIN 4.x (2)

Only saving the best solution.

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

```
c_state "int  min_cost" "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**-label of the **TSP** proctype (i.e. in the final state of a possible solution):

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

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

TSP in SPIN 4.x (3)

Optimisation: simple (branch &) bound.

Simple optimization:

- If after visiting a place, the **cost** is already greater than **min_cost**, we know that this execution trace will not lead to a better trace. So, we could stop searching the state space.

At the beginning of each place **Pi**:

```
Pi: atomic {
    vv[i] = true;
    if
    :: c_expr { now.cost > min_cost } -> goto end
    :: else
    fi;

    if
    :: !vv[1] -> cost = ... ; goto P1
    :: ...
    :: else   -> cost = ... ; goto end
    fi
}
```

executable if the C expression is non-zero

Beware: we are now changing the model.

TSP in SPIN 4.x (4)

Optimisation: (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 an LTL property (i.e. combination with original idea).

We *on-the-fly* check ◊ **(now.cost ≥ min_cost)**:

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

Big advantage:

- We do not have to change the model to prune the search tree.

By the way, please note that SPIN verifies a LTL formula using a never claim, that is automatically generated from the LTL formula.
- This means that we let SPIN use its liveness machinery to solve a safety problem.

Reordering the model

- 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

Nearest Neighbour

Note that changing the order of the guards does not change the behaviour of the model.

- Nearest Neighbour (NN) Heuristic
  - The salesman always goes to the nearest city (lowest cost), which has not yet been visited.
  - To apply the NN-heuristic with SPIN it is sufficient to sort the guards in the if-clauses according to the cost.

```
active proctype TSP() {
...
P7: atomic {
...
if
:: !vv[1] -> cost = cost+11; goto P1
:: !vv[2] -> cost = cost+ 9; goto P2
:: !vv[3] -> cost = cost+23; goto
:: !vv[4] -> cost = cost+21; goto
:: !vv[5] -> cost = cost+ 7; goto
:: !vv[6] -> cost = cost+14; goto P6
:: else    -> cost = cost+20; goto end
fi
}
...
```

```
active proctype TSP() {
...
P7: atomic {
...
if
:: !vv[5] -> cost = cost+ 7; goto P5
:: !vv[2] -> cost = cost+ 9; goto P2
:: !vv[1] -> cost = cost+11; goto P1
:: !vv[6] -> cost = cost+14; goto P6
:: !vv[4] -> cost = cost+21; goto P4
:: !vv[3] -> cost = cost+23; goto P3
:: else    -> cost = cost+20; goto end
fi
}
...
```


TSP - some experiments

PIII/Mobile 1Ghz 256 Mb
SPIN 4.0.1 Windows 2000

#states	N=11	N=12	N=13	N=14	N=15
no B&B	572729	1878490	5459480	o.m.	o.m.
unsorted BB in model	278753	212984	514332	2478440	2820880
unsorted BB in property	111920	72022	173309	1050580	1010080
sorted (NN) BB in model	132517	54924	140075	1748130	1388100
sorted (NN) BB in property	49801	16662	43240	737107	480572

Of course, we used some automated scripts to (i) generate random cost matrices, (ii) generate the Promela models from these matrices and (iii) run SPIN on these models.

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 `min_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 >= min_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 `min_cost` so far.

TSP - final remarks

- Observations
  - no need to check for acceptance cycles in never claim
    - will have impact on the search time (roughly divided by 2).
  - no need to store states
    - The search space is always a tree with a continuously growing cost on the paths: the DFS stack is enough to complete the search.

      `gcc -DBITSTATE -DRANDSTOR=-1 ...`
      - uses bitstate hashing
      - sets change of storing a state to less than -1%
    - Now we can find solutions for TSPs of arbitrary size (as long as there is enough time…).

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.x 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

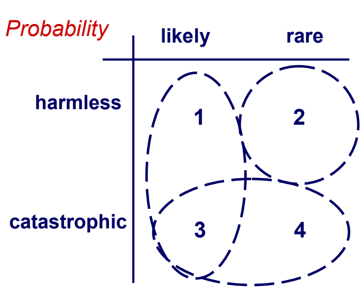
Part 1

- 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

Added value of logic verification



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

some additional documentation

The Spin Model Checker
Primer and Reference Manual

Gerard J. Holzmann

Addison Wesley
ISBN 0-32122-862-6, 608 pgs

Complete lecture notes from a course on Logic Model Checking based on the new book, taught at Caltech University, Jan-Mar. 2004 are available online at:

http://spinroot.com/spin/Doc/course/index.html

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 1,2 and repeat 3,4,5 until happy…

The One-Slide Theory

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

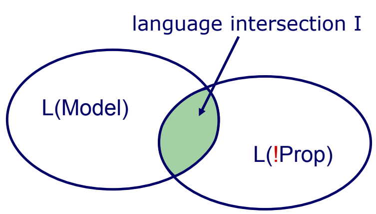
$$L(Model) \cap (\Sigma^{\omega} \setminus L(Prop)) = \varnothing$$

- i.e.:

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

logical negation of the property

Intersection of 2 formal languages

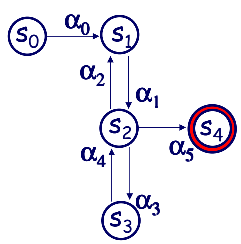


if I is empty:	the Model satisfies the Property
if I is non-empty:	the Model can violate the Property
	and I contains at least one counter-example

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
  - $T \subseteq S \times L \times S$ transition relation

An example



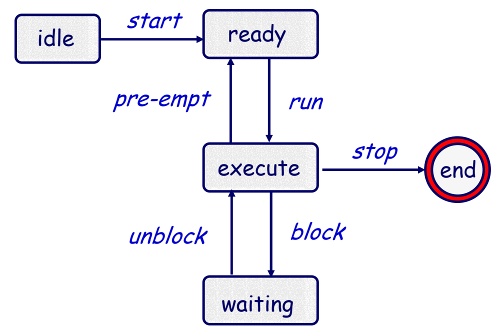
S: { s_0, s_1, s_2, s_3, s_4 }

L = { α_0, α_1, α_2, α_3, α_4, α_5 }

F = { s_4 }

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

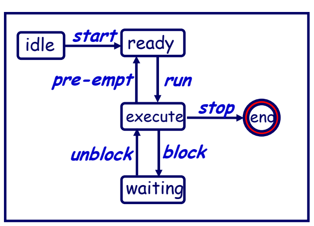
An interpretation



The definition of a run

a *run* of automaton {S, s_0, L, T, F} is an ordered set
σ = {s_0, s_1, s_2, ..., 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

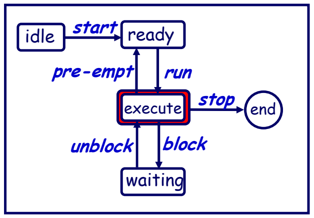
a finite run σ = {s_0, s_1, s_2, ..., s_k}
of automaton {S, s_0, L, T, F} is accepted if and only if its final state $s_k \in$ **F**.

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 if and only if
$\exists\, s_k \in \mathbf{F}$, s_k appears infinitely often in σ.

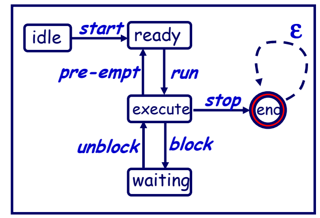


run: { idle, [ready, execute,]* }
word: {*start*, [*run*, *pre-empt*,]* }

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 formalism to state properties of concurrent systems
- Leading candidate: 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

Spin's LTL syntax

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: σ = { s_0, s_1, s_2, s_3 … }

propositional symbols: p,q,…
$\forall$i,(i ≥ 0) and $\forall$ p, s_i |= p is defined
temporal formulae: e,f,…

→ σ |= f iff s_0 |= f

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

Typical LTL formulae

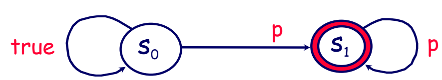
[]p	always p	invariance
<>p	eventually p	guarantee
p -> (<>q)	p implies eventually q	response
p -> (q U r)	p implies q until r	precedence
[]<>p	always, eventually p	recurrence (progress)
<>[]p	eventually, always p	stability (non-progress)
<>p -> <>q	eventually p imlies eventually q	correlation

useful equivalences:
![] p <=> <> !p
!<> p <=> [] !p

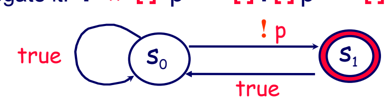
avoiding X:
next-time-free properties
are stutter-invariant

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)) = ∅

none of the strongly connected components in the reachability graph of the intersection of system and claim may contain an accepting state

(any cycle through an accepting state is 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');
}
```

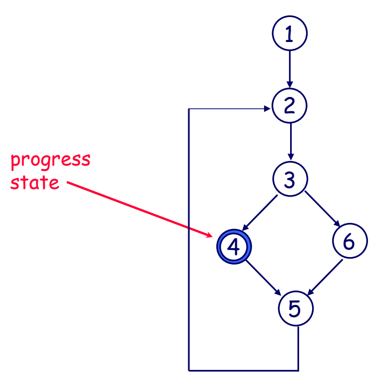
- recursively explore the state graph
- store as little data as possible
- no need to store transitions
- need not store all states
- in approximate searches, need not store states accurately

Cycle detection (1)

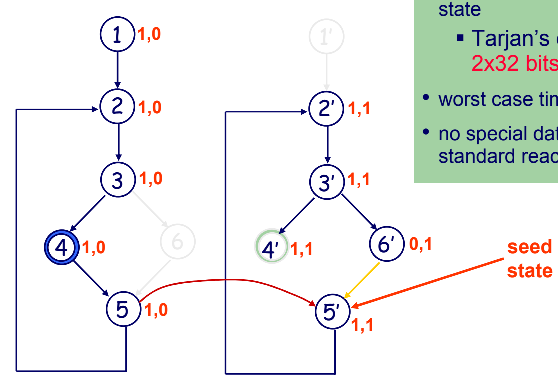
- errors (violations of an LTL property) correspond to runs with infinitely many accepting states
- in a finite graph, these correspond to reachable strongly connected components (scc's) with accepting states
- Tarjan's algorithm constructs the scc's in polynomial time -- can check each for the presence 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

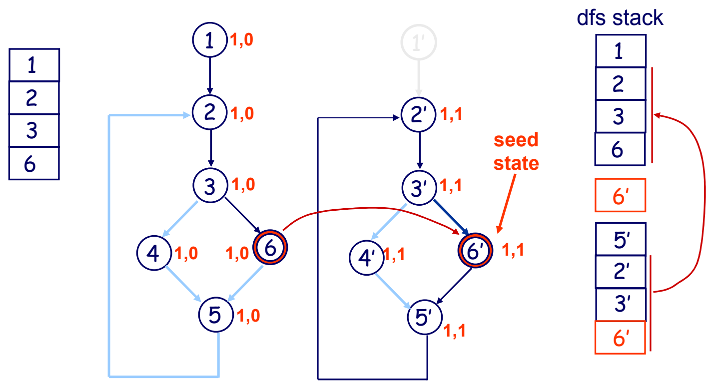


Nested depth-first search



- 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



Computational 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.

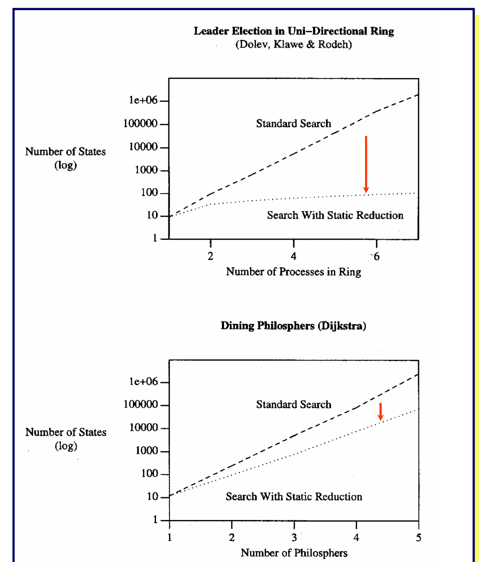
strategies for reducing S

- lossless memory compression
- don't store states but a minimized DFA recognizer
- lossy compression (proof approximation)

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

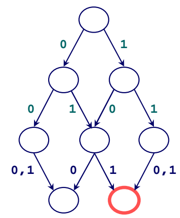
Effect of partial order reduction



Minimized dfa storage in spin

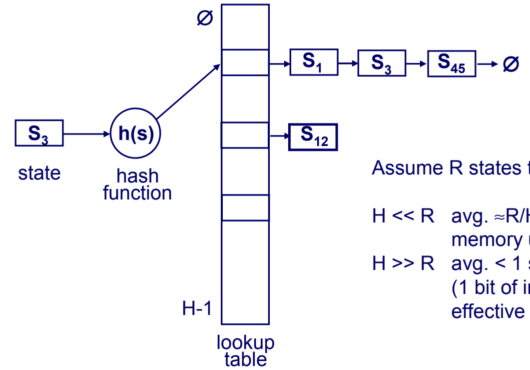
state descriptors can be stored as a
minimized deterministic finite automaton

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."* [Bob 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:

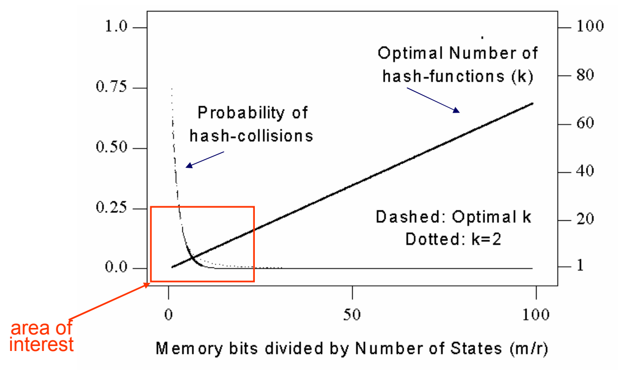
$$\left(1 - \frac{1}{m}\right)^{k.r}$$

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

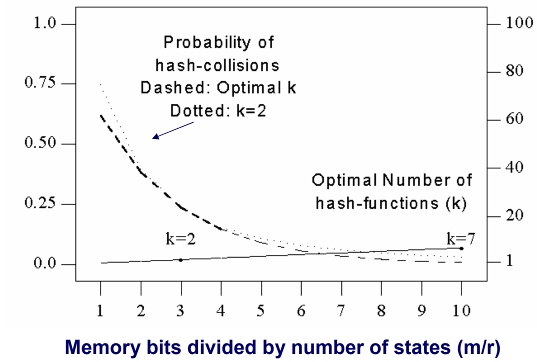
$$\left(1 - \left(1 - \frac{1}{m}\right)^{k.r}\right)^{k} = \left(1 - e^{-k.r/m}\right)^{k}$$

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

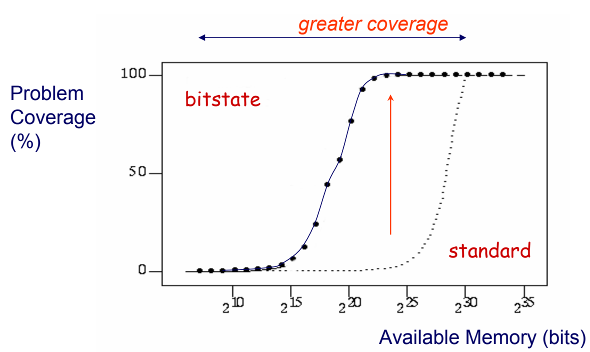
The optimal number of hash-functions



Close-up view



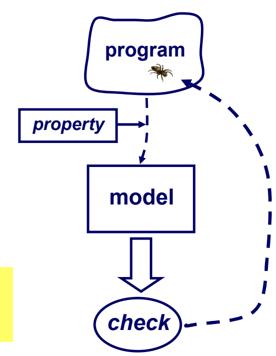
The effect of bitstate hashing



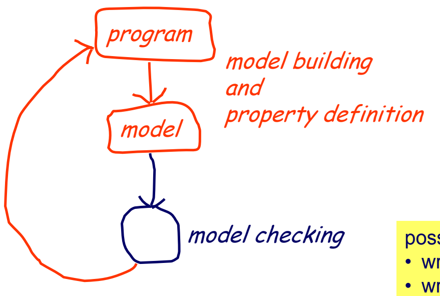
(Data: a Commercial Data Transfer Protocol)

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



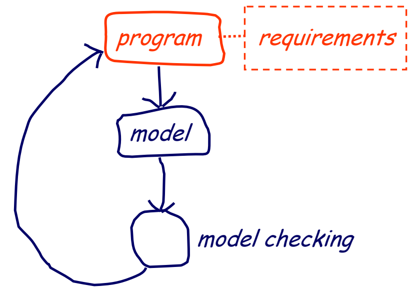
manual

automated

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



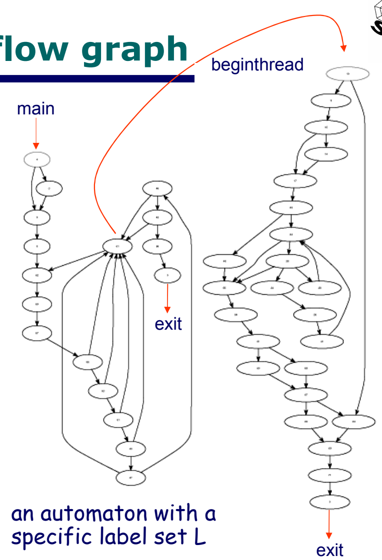
manual

automated

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

Program control flow graph

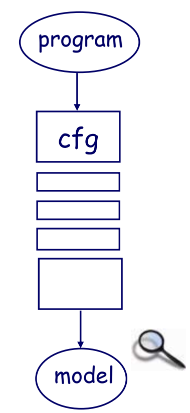
```
int
main(int argc, char *argv[])
{       char buf[512];
        int ns;

        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 (FeaVer)

- 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)