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.

Model Checking

\[
\phi = M_n < 3
\]

Property \( \phi \) is satisfied.

Model Checker

State Space

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 vs Modern Approach

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)
**Documentation on SPIN**

- SPIN's starting page: [http://spinroot.com](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/](http://spinroot.com/gerard/)
- SPIN version 1.0 is described in Holzmann 1991.

## Promela Model

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

## Promela statements

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

**Basic recipe to check**

1. **Sanity check**
   - Interactive and random simulations.
2. **Partial check**
   - Use SPIN's bitstate hashing mode to quickly sweep over the state space.
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)**
\( M \models \varphi \)

### Simulation Algorithm

\[
\text{while} \ (\neg \text{error} \land \neg \text{allBlocked}) \ do \\
\text{ActionList menu = getCurrentExecutableActions();} \\
\text{allBlocked = (menu.size() == 0);} \\
\text{if } \neg \text{allBlocked} \text{ then} \\
\text{Action act = menu.chooseRandom();} \\
\text{error = act.execute();} \\
\text{fi} \\
\text{od} \\
\]

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 not in Statespace then dfs(t), fi
end dfs
end dfs
```

- Only works for safety properties.

- The old states are stored on a stack, which corresponds with a complete execution path.

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

### Verification Algorithm (2)

```
S \quad \text{language intersection} \quad A

\neg \varphi \quad \text{translation} \quad \text{accepts words}

\neg \varphi \quad \text{Buchi Automaton}

\text{Procedure: search for an accepting state in the intersection, which is reachable from itself.}

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”
  - invariance: \( x \) is always less than 5
  - deadlock freedom: the system never reaches a state where no actions are possible
  - 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 \( n \times m \) bytes

SPIN provides several algorithms to compress the state vector.

SPIN’s Reduction Algorithms

- SPIN has several optimisation algorithms to make verification runs more effective:
  - partial order reduction
  - 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 these (default) optimisation/reduction algorithms.

Moore’s Law & Advanced Algorithms

- Verification results of Tpc (The phone company)
  - Available Memory
  - Required Memory

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

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

<table>
<thead>
<tr>
<th></th>
<th>BRP 1997</th>
<th>BRP 2002</th>
</tr>
</thead>
<tbody>
<tr>
<td>state vector</td>
<td>104 bytes</td>
<td>96 bytes</td>
</tr>
<tr>
<td># states</td>
<td>1,799,340</td>
<td>169,208</td>
</tr>
<tr>
<td>Memory (Mb)</td>
<td>116,399</td>
<td>14,354</td>
</tr>
</tbody>
</table>

Both verified with SPIN 3.4.x

took up to an hour in 1997

took 2 sec. in 2002

Art of (Promela) Modelling

- 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

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.

Space vs. time considerations, priorities:
1. Number of states
2. Size of the state vector
3. Maximum search depth
4. Verification time

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;
      stat2;
      ...
    }
    ```
  3. Check that aflag is always 0.
    ```
    []!aflag
    ```
  4. E.g. active process monitor:
    ```
    assert(!aflag);
    ```
Invariance

- $[] \neg P$ 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:
    - $[] \neg \text{aflag}$
    - $[] \neg \text{mutex} \neq 2$
- SPIN supports (at least) 7 ways to check for invariance.

variant 1+2 - monitor process (single assert)

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

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

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

variant 3 - guarded monitor process

- Drawback of solution "1+2 monitor process" is that the $\text{assert}$ statement is executable in every state.

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

- The $\text{atomic}$ statement only becomes executable when $P$ itself is not true.

variant 4 - monitor process (do assert)

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

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

- But the number of states is clearly advantageous.
variant 5 - never claim (do assert)

- 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

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

Invariance experiments (1)

- PII 330Mhz
- 128 Mb
- SPIN 3.3.10
- Linux 2.2.12
- NO partial order reduction

![Invariance experiments graph]

- 1. monitor first
- 2. monitor last
- 3. guarded monitor
- 4. monitor do assert
- 5. never do assert
- 6. LTL property
- 7. unless
Invariance experiments (2)

- DNOREDUCE
- NO partial order reduction

1. monitor first
2. monitor last
3. guarded monitor
4. monitor do assert
5. never do assert
6. LTL property
7. unless

Invariance experiments (3)

default settings
memory (Mb)

1. monitor first
2. monitor last
3. guarded monitor
4. monitor do assert
5. never do assert
6. LTL property

Invariance experiments (4)

default settings
time (sec)

1. monitor first
2. monitor last
3. guarded monitor
4. monitor do assert
5. never do assert
6. LTL property

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 timeouts.
- Variant 3 “guarded monitor process” is the most effective and reliable method for checking invariance.

seems attractive...
Invariance - Conclusions

• Generalizing, if one need to check \([P \land Q \land \ldots \land Z]\)
  one should use:

```promela
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 modeling choices
  - abstractions
  - identifying points of attention

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: \(\varphi_i\)
  - validation results:
    - simulation traces
    - verification results
  - directives and options to build verifiers
  - notes and remarks on validation runs

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

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

SPIN Verification Report

Property was satisfied

```
property was satisfied
```

the size of a single state

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

Full state space search for: never-claim 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
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
```

© Gerard J. Holzmann & Theo C. Ruys - Advanced SPIN Tutorial
Systematic Verification Model

- Design phase
  - System description
  - Requirements
  - Modeling
    - M, S
  - Simulation
    - M, S

Modeling phase

- Abstract
  - M, ϕ
  - Model check
    - Error
      - Interpret error
        - Correct error
        - Reverify models
      - Correct error
        - Model check
  - M, ϕ
  - All done?

Verification phase

- Verified M w.r.t. S

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

Beyond Xspin

- Personal” SPIN setup
  - Version control system
  - Literate programming tool

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

"Hippies" problem

- Germany
- Holland
- How fast?
- Holes
- Coffee shop
- "Stoned" hippies

Original “soldiers” problem (Ruys & Brinksma, 1998)
It is clear that we need to send two hippies over to Holland but only one hippie back with the flashlight.

channel germany_to_holland = [0] of {hippie, hippie};
channel holland_to_germany = [0] of {hippie};
channel stopwatch = [0] of {hippie};
byte time;

active proctype Germany() {
    bit here[N]; hippie h1, h2;
    do ::
        germany_to_holland ? h1, h2;
        here[h1] = 1; here[h2] = 1;
        stopwatch ? max(h1, h2);
        if all_gone -> break FI;
        select_hippie(h1);
      od
  }

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

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

#define max(x,y) ((x>y) -> x : y)
#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
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:
     - min = guess of (worst case, maximum) cost
     - do
       - verify M \[\triangleright\] (cost \[\geq\] min)
     - if (error)
       - min = cost
     - 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.

"eventually cost will be larger than min"

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 \(\triangleright\) (cost \[\geq\] 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 \[\geq\] min holds.

Scheduling with SPIN  (3)

- Good solution should be in the left part of the state space.
- Not all states are visited.
- Example: MAX INT > 0 > 0 > 0 > 0
- We iteratively check \(\triangleright\) (cost \[\geq\] min)
- The variable best_cost is initialised in a special section before the verification is started.

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` to use atomic C statements 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 \cite{Holzmann2000}.
  - "... 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.

**Hippies in SPIN 4.0 (1)**

```c
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 puttrail()`.

**Hippies in SPIN 4.0 (2)**

```c
(c_state "int best_time" "Hidden" "1000"

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

- This declaration is copied verbatim to `pan.c`.
- "Hidden" means that the variable is not stored in the state vector, but is global for the whole verification.
- Only saving the best schedule.
**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;
```

**In the Holland proctype:**
```
stopwatch ! max(h1,h2);
if :: c_expr { (now.time > best_time) } -> break
else;
```

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

**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).

```
#define higher_cost (c_expr { now.time >= best_time })
never {
/* !<> higher_cost */
accept_init:
T0_init:
if:: (! (higher_cost)) -> goto T0_initfi;
}
```

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

**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:

```c
<> 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.

---

**Verification results with SPIN 4.0**

<table>
<thead>
<tr>
<th>version</th>
<th>description</th>
<th># states</th>
</tr>
</thead>
<tbody>
<tr>
<td>original</td>
<td>deadlock checking</td>
<td>2541</td>
</tr>
<tr>
<td>original</td>
<td>0 (time ≥ 60)</td>
<td>1325</td>
</tr>
<tr>
<td>hippies 1</td>
<td>just print all schedules found</td>
<td>2659</td>
</tr>
<tr>
<td>hippies 2</td>
<td>saving the best schedule</td>
<td>2630</td>
</tr>
<tr>
<td>hippies 3</td>
<td>branch &amp; bound in processes</td>
<td>1766</td>
</tr>
<tr>
<td>hippies 4</td>
<td>branch &amp; bound in never claim</td>
<td>1330</td>
</tr>
</tbody>
</table>

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.
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
  add s to Statespace
  foreach successor t of s do
    if t not in Statespace then dfs(t)
  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); }
}
```

Changing the order of the guards

```
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);
Reordering the model (5)

changing the order of process creation

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: \( L(\text{Model}) \)
- Requirement: \( L(\text{Prop}) \)
- Show that: \( L(\text{Model}) \subseteq L(\text{Prop}) \)
- Method:
  \[
  L(\text{Model}) \cap (\Sigma^\omega \setminus L(\text{Prop})) = \emptyset
  \]
  i.e.:
  \[
  L(\text{Model}) \cap L(\neg \text{Prop}) = \emptyset
  \]

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
  - \( T \subseteq S \times L \times S \) transition relation

An example

- \( S = \{s_0, s_1, s_2, s_3, s_4\} \)
- \( L = \{\alpha_0, \alpha_1, \alpha_2, \alpha_3, \alpha_4, \alpha_5\} \)
- \( F = \{s_4\} \)
- \( T = \{(s_0, \alpha_0, s_1), (s_1, \alpha_1, s_2), ...\} \)
An interpretation

```
idle               ready
|                  v
| pre-empt   run
execute

unblock         block

waiting         end
```

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
\[
\forall i, 0 \leq i < k : \exists \alpha \in L \text{ and } (s_i, \alpha, s_{i+1}) \in T.
\]

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
\[
\exists s_k \in F, \text{ } s_k \text{ appears infinitely often in } \sigma.
\]

Language: set of all \(\omega\)-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 \(\varepsilon\)-symbol)

run: \{ idle, ready, execute, waiting, execute, [end,*] \}
word: \{start, run, block, unblock, stop, \(\varepsilon\)\}*

diagram:

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 \(\omega\)-automata

• Example:
  \[ (a \neq b) \rightarrow <> (a = b) \]
  it is always the case \[\] that when \((a \neq b)\) 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 \(p, q, r, \ldots\)
  \(f\)
  unary \(f\)
  binary \(f\)

unary ::= 
  \[\]  --- always, henceforth
  <>  --- eventually
  \(X\)  --- next
  !  --- logical negation

cautions

binary ::= 
  \(U\)  --- strong until
  \&\&  --- logical and
  | |  --- logical or
  \(\rightarrow\)  --- implication
  <>  --- equivalence

LTL semantics

run: \(\sigma = \{ s_1, s_2, s_3, \ldots \}\)

propositional symbols: \(p, q, \ldots\)

\(\forall i, (i \geq 0)\) and \(\forall p, s_i \models p\) is defined

\(\sigma \models f\) iff \(s_0 \models f\)

with:

\[ s_i \models f \] iff \(s_i \models f\)

\(\forall j, (j > i) : s_j \models f\) iff

\(s_i \models \top \) iff

\(\exists j, (j > i) : s_j \models f\) and

\(\forall k, (k < i) : s_k \models f\)
### Typical LTL Formulae

<table>
<thead>
<tr>
<th>Formula</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>[]p</td>
<td>always p</td>
</tr>
<tr>
<td>&lt;&gt;p</td>
<td>eventually p</td>
</tr>
<tr>
<td>p \rightarrow (&lt;&gt;q)</td>
<td>p implies eventually q</td>
</tr>
<tr>
<td>p \rightarrow (q \equiv r)</td>
<td>p implies q until r</td>
</tr>
<tr>
<td>[]&lt;&gt;p</td>
<td>always, eventually p</td>
</tr>
<tr>
<td>&lt;&gt;(!p)</td>
<td>eventually, always p</td>
</tr>
<tr>
<td>&lt;&gt;p \rightarrow &lt;&gt;q</td>
<td>eventually p implies eventually q</td>
</tr>
</tbody>
</table>

**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 \( \omega \)-runs for which \( f \) is satisfied
- Example: the formula \( <>[]p \) corresponds to the non-deterministic Büchi automaton:

```
true
```

```
S_0
\rightarrow
P
```

```
S_1
```

- To turn a property \( f \) into a claim (the complement of \( f \)), it suffices to negate it:

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

\equiv

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

### Automata Theoretic Verification

- System: \( L \)(system)
- Property: \( L \)(prop)
- Show: \( L \)(system) \subseteq L \)(prop)
- Or that: \( L \)(system) \cap (L^\omega \setminus L \)(prop)) = \emptyset

---

### Reachability by Depth-First Search

```
expose()
{
    store = {};
    dfs(s)
}
dfs(s)
{
    if s \in store
        return;
    else
        store = store \cup {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: \( S \)

\[ R \times S \] 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**

*Example:*

- states = \{ 011, 101, 110, 111 \}

*Minimized DFA storage in Spin*

- state descriptors can be stored as a minimized deterministic finite automaton

*Example:*

- \( s \) 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

S3 h(s) state hash function

S3 S1 S4 S5
lookup table

H-1

Assume R states to be stored

H << R avg. \( \approx \frac{R}{H} \) states/slot

memory use \( R.S + \text{overhead} \)

H >> R avg. < 1 state/slot

(1 bit of information/slot)

effective memory use \( R \) bits

Robert Morris 1968

- assume \( H \gg R \), 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.”

\[ \text{Morris, CACM1968} \]

- 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:

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

\( \text{rhs is minimized for } k = \ln 2 \times \frac{m}{r} \)

The optimal number of hash-functions
**Close-up view**

*Probability of hash-collisions*
- Dashed: Optimal \( k \)
- Dotted: \( k=2 \)

*Optimal Number of hash-functions (k)*

*Memory bits divided by number of states (m/r)*

**The effect of bitstate hashing**

*Problem Coverage (%)*

*Available Memory (bits)*

(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

**Classic model checking**

*model building and property definition*

*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

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

Program control flow graph

Syntactic conversion

```c
int c, nl, nw, nc;
int inword = 0;

main(void)
{  
    printf("%d\t%2d\t%2d\n", nl, nw, nc);
}
```

```Promela
C

int c, nl, nw, nc;

main(void)
{   
    printf("%d\t%2d\t%2d\n", nl, nw, nc);
}
```


```
int c, nw, nl, nc;
init {
    bool inword = false;
    do
        STDIN?c -> nc++;
        if :: c == '
' -> nl++
        :: else
        fi;
        if :: c == ' ' || c == '
' ->
            inword = false
        :: else ->
            if :: inword == false ->
                nw++; inword = true
            :: else /* do nothing */
            fi
        fi
    od;
    printf("%d	%d	%d\n", nl, nw, nc)
}
```

**Slicing**

- Property: \[ (nl \geq nc) \]
- Slice criteria: \{nl, nc\}

**Added value of logic verification**

<table>
<thead>
<tr>
<th>Damage</th>
<th>Probability</th>
</tr>
</thead>
<tbody>
<tr>
<td>likely</td>
<td>rare</td>
</tr>
<tr>
<td>harmless</td>
<td>1 2</td>
</tr>
<tr>
<td>catastrophic</td>
<td>3 4</td>
</tr>
</tbody>
</table>

- 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