SPIN

SPIN (= Simple Promela Interpreter)

= is a tool for analysing the logical consistency of concurrent systems, specifically of data communication protocols.

= state-of-the-art model checker, used by >5000 users

- concurrent systems are described in the modelling language called Promela.

Documentation

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

Available in PDF format from www.spinroot.com.

New book on SPIN (Sep. 2003):
Gerard J. Holzmann
The Spin Model Checker
Primer and Reference Manual
Addison Wesley
ISBN 0-32122-862-6, 608 pages.
describes SPIN up to version 4.0.

Promela

Promela (= Protocol/Process Meta Language)

- specification language to model finite-state systems
- loosely based on CSP
  - dynamic creation of concurrent processes
  - communication via message channels can be
    - synchronous (i.e. rendezvous), or
    - asynchronous (i.e. buffered)
- features from Dijkstra's guarded command language
- features from the programming language C

Promela is a modelling language,
not a programming language!

Promela Model

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

type declarations: `mtype`, constants, typedefs (records)

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

global variable declarations:
- simple vars
- structured vars
- vars can be accessed by all processes

process declarations: behaviour of the processes: local variables + statements

[init process]: initialises variables and starts processes

Process

- A process type (`proctype`) consist of
  - a name
  - a list of formal parameters
  - local variable declarations
  - body

```
proctype Sender(chan in; chan out) {
        bit sndB, rcvB;
        do
        :: out ! MSG, sndB ->
               in ? ACK, rcvB;
               if
               :: sndB == rcvB -> sndB = 1-sndB
               :: else -> skip
               fi
        od
}
```

name
formal parameters
local variables
body

The body consists of a sequence of statements.

Statements

Statements are separated by a semi-colon: ";".
or by the equivalent "->"

- The body of a process consists of a sequence of statements. A statement is either
  - executable: the statement can be executed immediately.
  - blocked: the statement cannot be executed.

executable/blocked depends on the global state of the system.

- An assignment is always executable.
- An expression is also a statement; it is executable if it evaluates to non-zero.

`2 < 3`	always executable
`x < 27`	only executable if value of `x` is smaller 27
`3 + x`	executable if `x` is not equal to –3

Statements

- `assert(<expr>);`
  - The `assert`-statement is always executable.
  - If `<expr>` evaluates to zero, SPIN will exit with an error, as the `<expr>` "has been violated".
  - The `assert`-statement is often used within Promela models, to check whether certain properties are valid in a state.

```
proctype monitor() {
  assert(n <= 3);
}

proctype receiver() {
  byte msg;
  ...
  toReceiver ? msg;
  assert(msg != ERROR);
  ...
}
```

We will later see that this is not the preferred way to check for invariance.

Mutual Exclusion

- Mutual Exclusion is a safety property among processes:

  Two processes may not interleave certain (sub-) sequences of instructions (i.e. the critical section). Instead, one sequence must be completed before the other commences.

- Properties to be satisfied:
  - at most one process can be in the critical section at the same time;
  - no deadlocks
  - no halting in the critical section
  - no starvation of one of the processes
- Java has a special construct to protect critical sections:
  - `synchronized`

Mutual Exclusion WRONG

```
bit  flag;   /* signals entering/leaving the section  */
byte mutex;  /* # procs in the critical section (CS). */

proctype P(bit i) {
  flag != 1;
  flag  = 1;
  mutex++;
  printf("MSC: P(%d) has entered section.\n", i);
  mutex--;
  flag  = 0;
}

proctype monitor() {
  assert(mutex != 2);
}

init {
  atomic { run P(0); run P(1); run monitor(); }
}
```

models:
`while (flag == 1) /* wait */;`

CS

Problem: assertion violation! Both processes can pass the `flag != 1` "at the same time", i.e. before `flag` is set to `1`.

starts two instances of process `P`

Mutual Exclusion

```
bit  a, b;    /* signal entering/leaving the section */
byte mutex;   /* # of procs in the critical section. */
byte turn;    /* who's turn is it?                   */

active proctype A() {
  a = 1;
  turn = B_TURN;
  b == 0 ||
    (turn == A_TURN);
  mutex++;
  mutex--;
  a = 0;
}

active proctype B() {
  b = 1;
  turn = A_TURN;
  a == 0 ||
    (turn == B_TURN);
  mutex++;
  mutex--;
  b = 0;
}

active proctype monitor() {
  assert(mutex != 2);
}
```

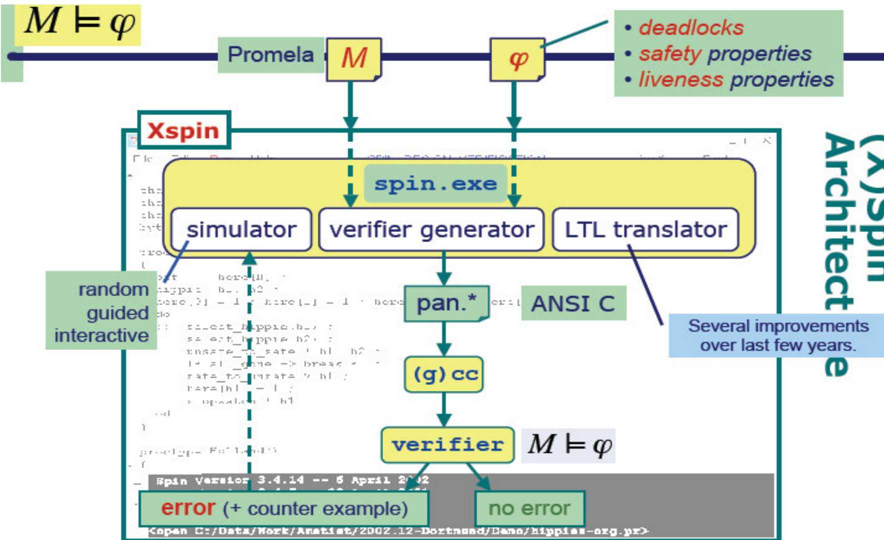
Either `B` does not yet want to enter, or `A` was here first.

Can be generalised to a single process.

Dekker [1962]

First "software-only" solution to the mutex problem (for two processes).

XSPIN



Important constructs

most important Promela constructs are either executable or blocked

	Construct	Executability
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

IF Statement

```
if
:: choice1 -> stat1.1; stat1.2; stat1.3; …
:: choice2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choicen -> statn.1; statn.2; statn.3; …
fi;
```

- If there is at least one `choice`$_i$ (guard) executable, the `if`-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no `choice`$_i$ is executable, the `if`-statement is blocked.
- The operator "`->`" is equivalent to "`;`". By convention, it is used within `if`-statements to separate the guards from the statements that follow the guards.

Do Statement

```
do
:: choice1 -> stat1.1; stat1.2; stat1.3; …
:: choice2 -> stat2.1; stat2.2; stat2.3; …
:: …
:: choicen -> statn.1; statn.2; statn.3; …
od;
```

- With respect to the choices, a `do`-statement behaves in the same way as an `if`-statement.
- However, instead of ending the statement at the end of the chosen list of statements, a `do`-statement repeats the choice selection.
- The (always executable) `break` statement exits a `do`-loop statement and transfers control to the end of the loop.

Example

- Example – modelling a traffic light

if- and do-statements are ordinary Promela statements; so they can be nested.

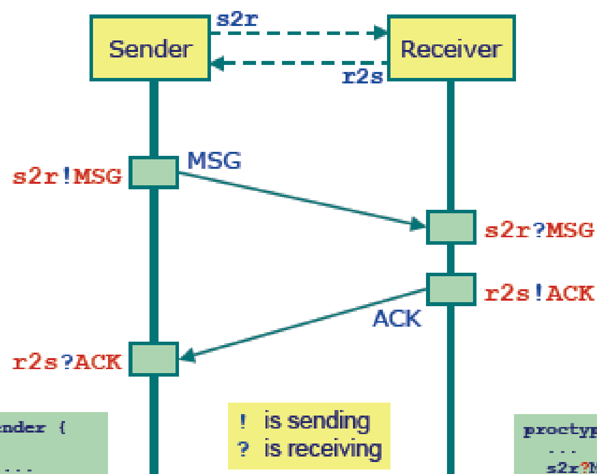
```
mtype = { RED, YELLOW, GREEN } ;

active proctype TrafficLight() {
    mtype state = GREEN;
    do
    ::  (state == GREEN)  -> state = YELLOW;
    ::  (state == YELLOW) -> state = RED;
    ::  (state == RED)    -> state = GREEN;
    od;
}
```

mtype (message type) models enumerations in Promela

Note: this do-loop does not contain any non-deterministic choice.

Communication 1




```
proctype Sender {
  ...
  s2r!MSG; ...
  r2s?ACK; ...
}
```

```
proctype Receiver {
  ...
  s2r?MSG; ...
  r2s!ACK; ...
}
```

Communication

- Communication between processes is via channels:
  - message passing
  - rendez-vous synchronisation (handshake)
- Both are defined as channels: — also called: queue or buffer

```
chan <name> = [<cap>] of {<t1>,<t2>, … <tn>};
```

- name of the channel: `<name>`
- type of the elements that will be transmitted over the channel: `{<t1>,<t2>, … <tn>}`
- number of elements in the channel: `<cap>` — `cap==0` is special case: rendez-vous

```
chan ch      = [1] of {bit};
chan toR     = [2] of {mtype, bit};
chan line[2] = [1] of {mtype, Msg};
```

`line[2]` — array of channels

Communication

- channel = FIFO-buffer (for `cap>0`)

! sending - *putting a message into a channel*

```
ch ! <expr1>, <expr2>, … <exprn>;
```

- The values of `<expri>` should correspond with the types of the channel declaration.
- A send-statement is executable if the channel is not full.

? receiving - *getting a message out of a channel*

message passing:

```
ch ? <var1>, <var2>, … <varn>;
```

- If the channel is not empty, the message is fetched from the channel and the individual parts of the message are stored into the `<vari>`s.

message testing:

```
ch ? <const1>, <const2>, … <constn>;
```

- If the channel is not empty and the message at the front of the channel evaluates to the individual `<consti>`, the statement is executable and the message is removed from the channel.

<var> + <const> can be mixed

```
ch!2,27    ch?x,y    ch?2,z    ch?2,2
```

`ch?2,2` — will be blocked

Communication

- Rendez-vous communication

  This is the only construct in Promela where two processes perform statements at the same time *(more or less)*.

  `<cap> == 0`

  The number of elements in the channel is now zero.
  - If send `ch!` is enabled and if there is a corresponding receive `ch?` that can be executed simultaneously and the constants match, then both statements are enabled.
  - Both statements will "handshake" and together take the transition.
- *Example:*

  ```
  chan ch = [0] of {bit, byte};
  ```
  - P wants to do `ch ! 1, 3+7`
  - Q wants to do `ch ? 1, x`
  - Then after the communication, `x` will have the value `10`.

Atomic

`atomic { stat`$_1$`; stat`$_2$`; ... stat`$_n$` }`

- can be used to group statements into an atomic sequence; all statements are executed in a single step (no interleaving with statements of other processes)
- is executable if `stat`$_1$ is executable
- if a `stat`$_i$ (with `i>1`) is blocked, the "atomicity token" is (temporarily) lost and other processes may do a step

- (Hardware) solution to the mutual exclusion problem:

```
proctype P(bit i) {
  atomic {flag != 1; flag = 1; }
  mutex++;
  mutex--;
  flag  = 0;
}
```

Process automaton

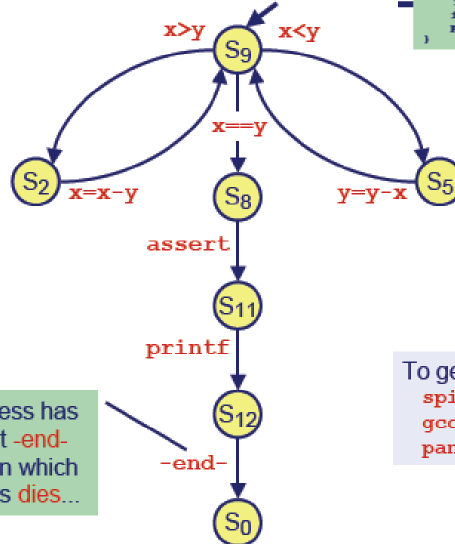
- Every promela `proctype` defines a finite state automaton, (S, s0, L, T, F), where
  - S   is a set of states
  - s0   is the initial state, s0 ∈ S
  - L   is a finite set of labels
  - T   is a set of transitions, T ⊆ S x L x S
  - F   is a set of final states, F ⊆ S

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.

Automaton

```
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...

Automata can also be viewed within Xspin (see “Run” menu).

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

-o3 is to disable statement merging

Interleaving semantic

- Promela processes execute concurrently.
- Non-deterministic scheduling of the processes.
- Processes are interleaved (statements of different processes do not occur at the same time).
  - exception: rendez-vous communication.
- All statements are atomic; each statement is executed without interleaving with other processes.
- Each process may have several different possible actions enabled at each point of execution.
  - only one choice is made, non-deterministically.

≈ randomly

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
```

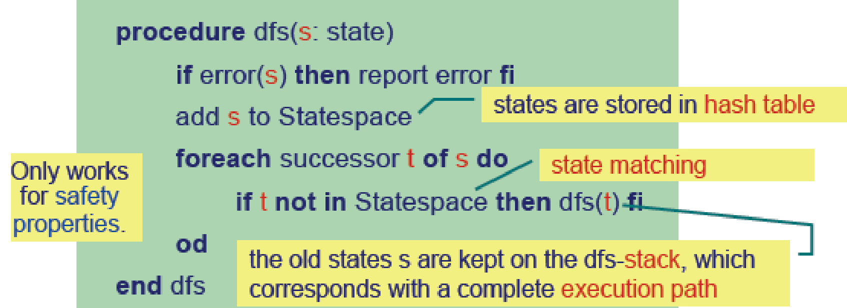
interactive simulation: act is chosen by the user

act is executed and the system enters a new state

Visit all processes and collect all executable actions .

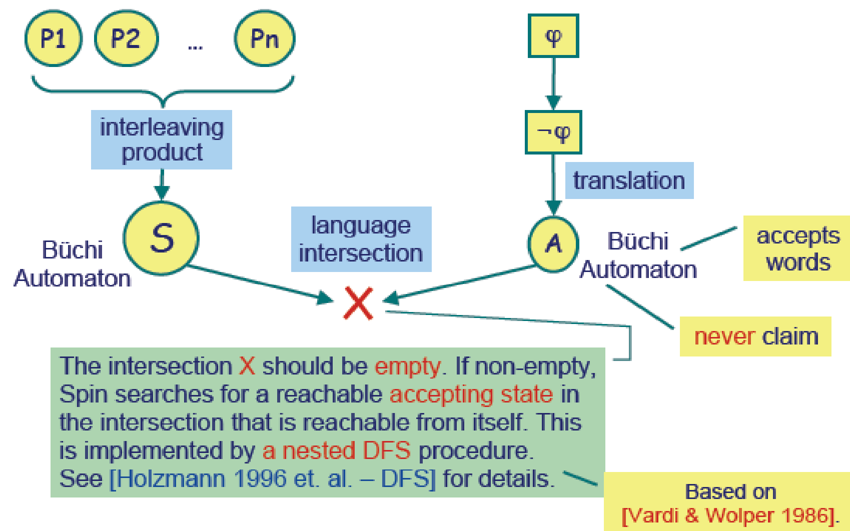
Verification Algorithm

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



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

Verification



State explosion problem

Sequential programs

- The # states of a program graph is:

$$|\,\#\text{program locations}\,| \cdot \prod_{variable\ x} |\,dom(x)\,|$$

$\Rightarrow$ number of states grows *exponentially* in the number of program variables
- N variables with k possible values each yields k^N states
- this is called *the state explosion problem*

- A program with 10 locations, 3 bools, 5 integers (in range $0 \ldots 9$):

$$10 \cdot 2^3 \cdot 10^5 = 800,000 \text{ states}$$

- Adding a single 50-positions bit-array yields $800,000 \cdot 2^{50}$ states

State explosion

Concurrent programs

- The # states of $P \equiv P_1 \| \ldots \| P_n$ is maximally:

$$\#\text{states of } P_1 \times \ldots \times \#\text{states of } P_n$$

$\Rightarrow$ # states grows *exponentially* with the number of components

- The composition of N components of size k each yields k^N states
- This is called *the state-space explosion problem*

Optimizations

- 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.

Partial Order Reduction

- Software model checking is prone to state explosion
- Two main causes
  - input nondeterminism
  - scheduling nondeterminism
- Example:

```
Thread 1        Thread 2            Thread n
  x1 := 1         x2 := 1    . . .    xn := 1
  stop            stop                stop
```

  - naive model checking: n! interleavings, 2^n states

- Partial-order reduction
  - explores subset of the state space, without sacrificing soundness

Partial order reduction

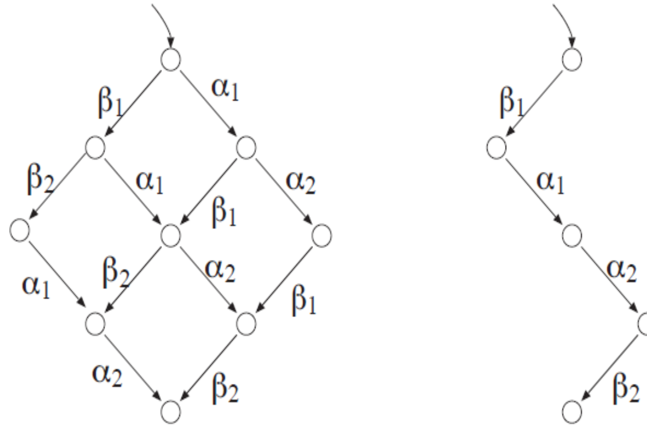
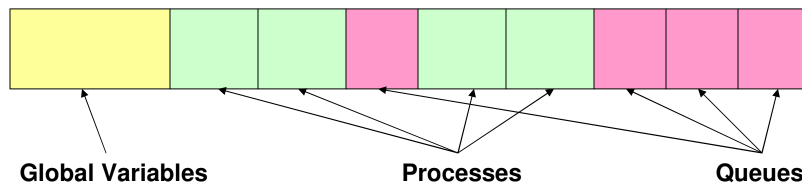


Figure 8.2: Parallel execution $(\alpha_1 ; \alpha_2) ||| (\beta_1 ; \beta_2)$.

State Representation - Compression

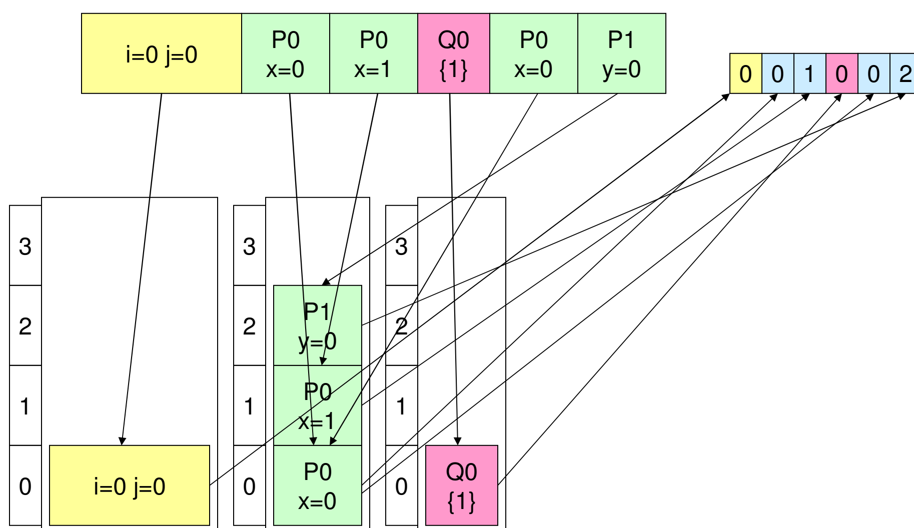
- Global variables
- Processes and local variables
- Queues



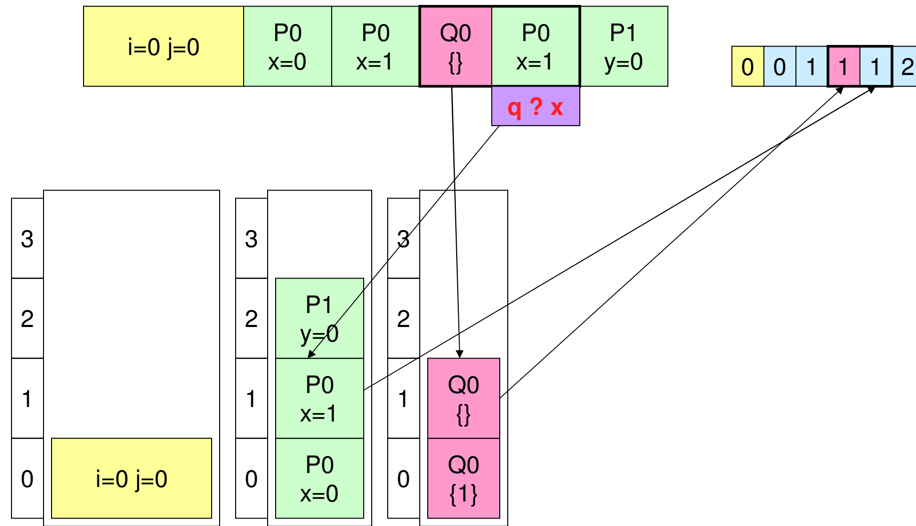
Compression

- Each transition changes only a small part of the state
- Assign a code to each element dynamically
- Encoded states + basic elements use considerably less spaces than the uncompressed states

Compression



Compression

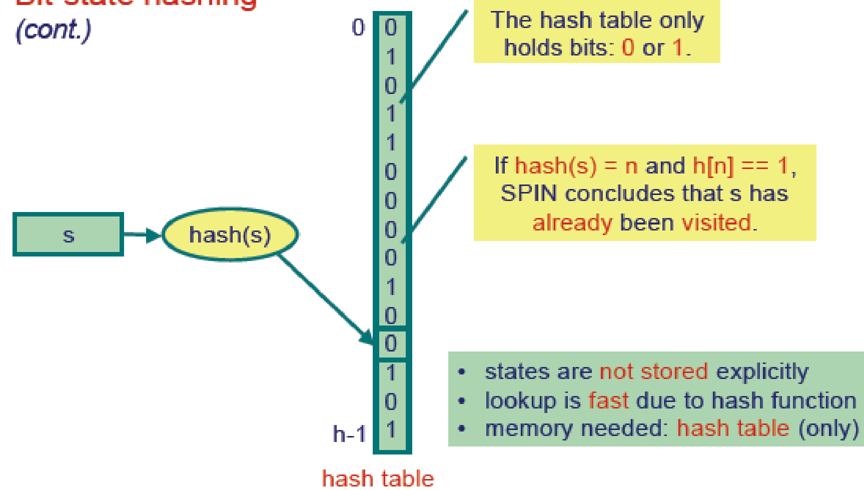


Bit State hashing

- Bit-state hashing [Holzmann 1998 – Bitstate hashing]
  - instead of storing each state explicitly, only one bit of memory is used to store a reachable state
  - given a state, a hash function is used to compute the address of the bit in the hash table
  - no collision detection
  - hash factor = # available bits / # reached states
    - aim for hash factor > 100
- Hash-compaction [Holzmann 1998 – Bitstate hashing]
  - large hash table: 2^64
  - store address in regular (smaller) hash table
  - with collision detection

Example

- Bit-state hashing
  (cont.)



NuSMV

What is NuSMV?

- Jointly developed at
  - **Center for Scientific and Technological Research** (ITC-IRST, Trento, Italy),
  - **Carnegie Mellon University** (Pittsburgh, USA).
- Model Checker for LTL and CTL.
- Supports Bounded Model Checking as well as Symbolic Model Checking.
- Verification of finite state systems described via Kripke structures.
- Modular approach: supports reusability.
- Specification of synchronous and asynchronous programs.

NuSMV

Differences between SPIN and NuSMV

Property	SPIN	NuSMV
Technology	explicit state	symbolic
Temporal Logic	LTL	LTL, CTL
System description	High Level language PROMELA (C-like)	Description of transition system
Verification	assertions, LTL, fairness, invariants	LTL, CTL, fairness, invariants

NuSMV

Why use NuSMV?

- Additionally supports CTL.
- Symbolic as well as bounded model checking possible: tackles state explosion problem.
- Closer to system, more concrete system description.

Why use SPIN?

- High level modelling language: supports embedded C code.
- Supports use of assertions.
- Easier to understand, easier to specify.
- Closer to programming language, more abstract system description.

Example

Simple Example

```
MODULE main
VAR
   request :  boolean;
   state   :  { ready, busy } ;
ASSIGN
   init(state) := ready ;
   next(state) :=
     case
      state = ready & request:  busy ;
      1                      :  { ready, busy } ;
     esac;
```

Modules

- One module has to be named *main*, additional modules possible.
- Modules may use parameters.

Example

Simple Example

```
MODULE main
VAR
   request :  boolean;
   state   :  { ready, busy } ;
ASSIGN
   init(state) := ready ;
   next(state) :=
     case
      state = ready & request:  busy ;
      1                      :  { ready, busy } ;
     esac;
```

State Variables

- Declaration of state variables.
- Only finite data types.

Example

Simple Example

```
MODULE main
VAR
   request :  boolean;
   state   :  { ready, busy } ;
ASSIGN
   init(state) := ready ;
   next(state) :=
     case
      state = ready & request:  busy ;
      1                      :  { ready, busy } ;
     esac;
```

Initial States

- Set of initial states possible!
- Initially, $state = ready$, *request* **unspecified**!

Example

Simple Example

```
MODULE main
VAR
   request :  boolean;
   state   :  { ready, busy } ;
ASSIGN
   init(state) := ready ;
   next(state) :=
     case
      state = ready & request:  busy ;
      1                      :  { ready, busy } ;
     esac;
```

Transition Relation

- Again: set of successor states possible.
- No constraints on *request* at all!

Synchronous vs. Asynchronous Systems

Synchronous Systems

- One MAIN module, possibly referring to other modules
- All modules run in **parallel**
- Assignments executed simultaneously.

Asynchronous Systems

- For asynchronous systems: use **processes**.
- Actions are **interleaved**.
- Only one process is running at a time.

Verification

NuSMV and LTL

- NuSMV allows specifications in LTL and CTL.
- Use
  LTLSPEC < *LTL specification* >
  SPEC < *CTL specification* >
  in your code.
- We concentrate on LTL.

Example

```
MODULE main
VAR
   gate1 :  inverter(gate2.output);
   gate2 :  inverter(gate1.output);
LTLSPEC
   (G F gate1.output & G F !gate1.output)
```

Extended LTL

- O p (read "once p "), stating that a certain condition p holds in one of the past time instants;
- H p (read "historically p "), stating that a certain condition p holds in all previous time instants;
- p S q (read "p since q "), stating that condition p holds since a previous state where condition q holds;
- Y p (read "yesterday p "), stating that condition p holds in the previous time instant.

Past temporal operators can be combined with future temporal operators, and allow for the compact characterization of complex properties.

A detailed description of the syntax of LTL formulas can be found in the NuSMV 2.3 User Manual.

Symbolic Model Checking

- State Space Explosion Problem
- Reduce memory requirement by utilizing compact representations of states/transitions
  - Boolean formulas represent sets and relations
  - Use fixed point characterizations of CTL operators

OBDD

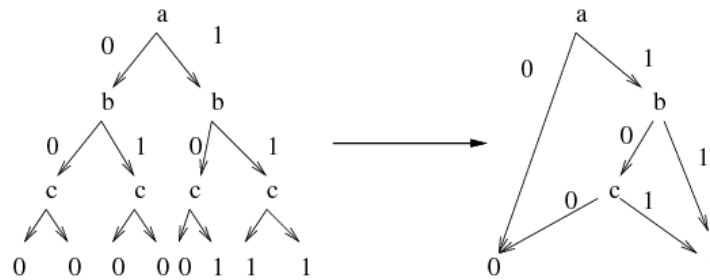


Figure 1. OBDD **representation for** $a \wedge (b \vee c)$.

Representation for States

- States as Boolean Formulas
  - 2^m states encoded by m proposition variables
  - State - conjunction of proposition or negative proposition
  - Set of States – conjunction of state (encoding) formula

Example: m = 2, $S=\{s_1,s_2,s_3,s_4\}$
  - Proposition Variables {a, b}
  - S={00, 01, 10, 11}=$\{\neg a \wedge \neg b, \neg a \wedge b, a \wedge \neg b, a \wedge b\}$
  - $\{s_1,s_2\}$={00, 01}=$(\neg a \wedge \neg b) \vee (\neg a \wedge b)$

Representation for Transitions

- Transitions as Boolean Formulas
  - (s, s’) encoded by two sets of proposition variables
  - Transition – conjunction of s and s’
  - Set of Transitions – conjunction of transition (encoding) formula

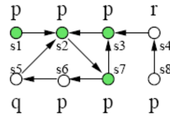
Example

$(s_4, s_3) = (11, 10) = a \wedge b \wedge a' \wedge \neg b'$

Symbolic Model Checking

- Atomic Propositions
  - $ROBDD(p) = \{s \mid p \in L(s)\}$
- $ROBDD(\neg p)$ = reversion of $ROBDD(p)$
- $ROBDD(p * q) = ROBDD\ (p) * ROBDD(q)$
  - $*$ is $\wedge$ or $\vee$
- $ROBDD(EXp(v)) = \exists v':[p(v') \wedge R(v, v')]$
- $(E(pUq)) = \mu Z.[q \vee (p \wedge EX\ Z)]$ – least fixpoint
- $(EGp) = \nu Z.[p \wedge EX\ Z]$ – greatest fixpoint

Example – greatest fixpoint

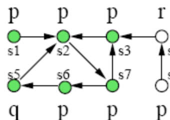


- To check: $\mathbf{EG}\, p$
- Compute: $\nu Z.\, p \wedge \mathbf{EX}\, Z$ (with Gfp)

$$
\begin{aligned}
Z_0 &= \mathit{True} \\
&= \{s_1, s_2, s_3, s_4, s_5, s_6, s_7, s_8\} \\
Z_1 &= p \wedge \mathbf{EX}\, Z_0 \\
&= \{s_1, s_2, s_3, s_6, s_7, s_8\} \\
Z_2 &= p \wedge \mathbf{EX}\, Z_1 \\
&= \{s_1, s_2, s_3, s_7\} \\
Z_3 &= p \wedge \mathbf{EX}\, Z_2 \\
&= \{s_1, s_2, s_3, s_7\}
\end{aligned}
$$

$Z_2 = Z_3$, so this is the greatest fixed point.

Example – least fixpoint



- To check: $\mathbf{E}p\,\mathbf{U}\,q$
- Compute: $\mu Z.\, q \vee (p \wedge \mathbf{EX}\, Z)$ (with Lfp)

$$
\begin{aligned}
Z_0 &= \mathit{False} &&= \emptyset \\
Z_1 &= q \vee (p \wedge \mathbf{EX}\, Z_0) &&= \{s_5\} \\
Z_2 &= q \vee (p \wedge \mathbf{EX}\, Z_1) &&= \{s_5, s_6\} \\
Z_3 &= q \vee (p \wedge \mathbf{EX}\, Z_2) &&= \{s_5, s_6, s_7\} \\
Z_4 &= q \vee (p \wedge \mathbf{EX}\, Z_3) &&= \{s_2, s_5, s_6, s_7\} \\
Z_5 &= q \vee (p \wedge \mathbf{EX}\, Z_4) &&= \{s_1, s_2, s_3, s_5, s_6, s_7\} \\
Z_6 &= q \vee (p \wedge \mathbf{EX}\, Z_5) &&= \{s_1, s_2, s_3, s_5, s_6, s_7\}
\end{aligned}
$$

$Z_5 = Z_6$, so this is the least fixed point.

Notes

- ROBDDs have pushed the limit of model checking from hundreds of variables to thousands.
- ROBDDs are sensitive to variable ordering (NP hard to find good ordering).
- ROBDDs can have exponential space requirements in worst case.
  - Combining N Counters with all possible transitions ➔ Reachability has to create N BDDs before it stabilizes.

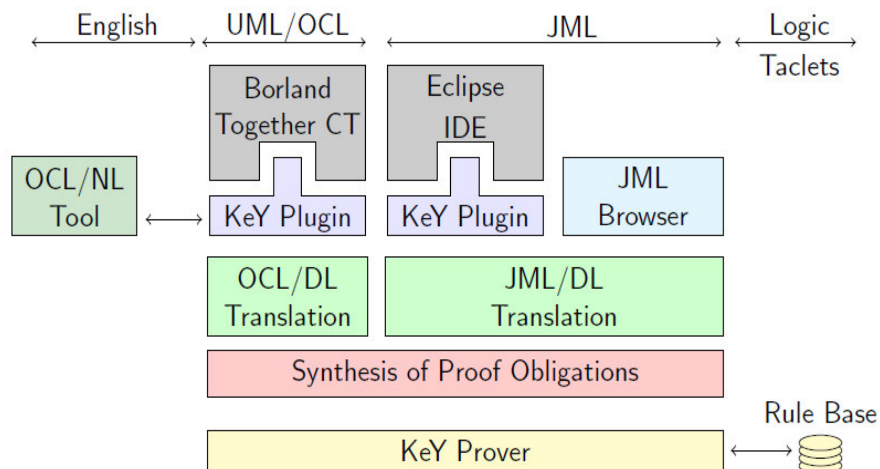
Other Approaches

- Prove(r) based system
  - KeY
- Timed Automata
  - UPPAAL
  - HyTech
  - Kronos

KeY Project

- Java Card as target language
- Integration with two standard SWE tools:
  - Borland Together, a commercial CASE tool
  - Eclipse, an open extensible IDE
- Specification languages
  - JML
  - UML/OCL
- Dynamic logic as program logic
- Verification = symbolic execution + induction
- Sequent style calculus + meta variables + incremental closure
- Interactive/automated prover with advanced UI

Key Architecture



Logic

Syntax

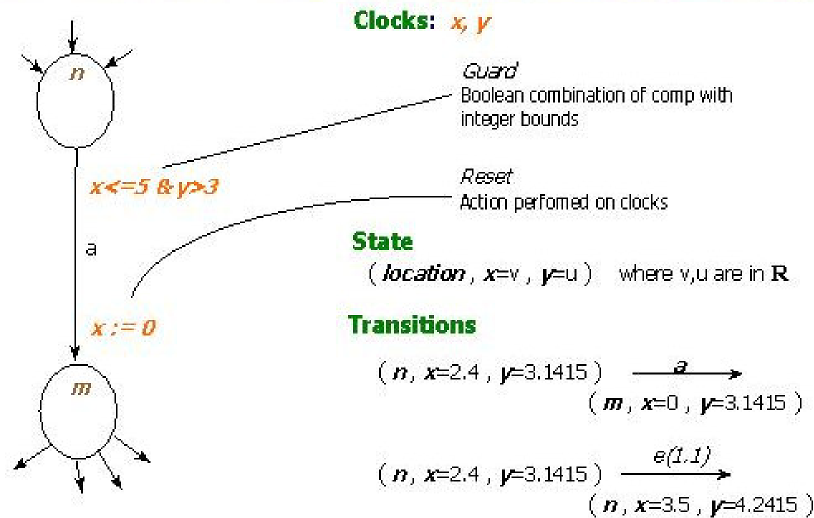
- Basis: Typed first-order predicate logic
- Modal operators $\langle p \rangle$ and $[p]$ for each (Java Card) program p
- Class definitions in background (not shown in formulas)

Semantics

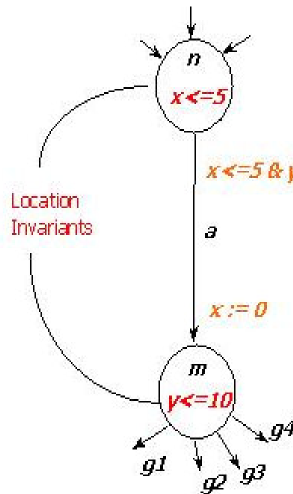
- Operators refer to the final state of p
- $[p]\ F$: If p terminates, then F holds in the final state (partial correctness)
- $<p>\ F$: p terminates and F holds in the final state (total correctness)

Java Card DL formulas contain unaltered Java Card source code

Timed Automata



Timed Automata



Clocks: x, y

Transitions

$(n, x=2.4, y=3.1415) \xrightarrow{e(3.2)}$ (crossed out)

$(n, x=2.4, y=3.1415) \xrightarrow{e(1.1)} (n, x=3.5, y=4.2415)$

Invariants insure progress!!

Notes

- UPPAAL allows for modeling time but is still based on CTL. Main drawbacks as for all explicit state model checkers. Integration with UML State charts was done.
- Kronos supports TCTL and therefore integrates time explicitly into logical properties/expressions.
- HyTech support linear hybrid automata. Has its own Logic. Allows the user to build himself a symbolic model checking computation.

Summary

- Motivation for Verification of Software
  - Software may be faulty -> ERROR
  - Spec is not complete ->unintended behaviors
  - In critical systems (Avionic, Medicine, …) errors have dangerous consequences (death, loosing huge amount of money,…).
- Verification of real systems
  - Hard to do but would be the best solution
  - Some approaches try to verify some important properties

Foundations for Verifying Systems

- Learned
  - Propositional Logic and Predicate Logic
    - Well Formed Formula
    - Semantics
    - Proof
  - CTL, LTL to include time aspects
    - Discussed systems with logical extensions
      - Alloy
      - TLA
      - SPIN

Approaches of Checking

- Transitions Systems form the base of most of the Checking approaches
- Explicit State checkers have to deal with the state explosion problem
  - Partial Order, State Compression, …
- Proof based checking can not be done fully automatic for a large set of models (only semi-automatic). Huge experience in how to write proofs (when, what rules to apply).

Notes

- Verification can help to increase the quality of specification and to check their "correctness" in terms of some properties.
- Verification can reduce the amount of testing for the target system
- Verification can also generate important Tests automatically (not discussed in WS). See "*Software Reliability Methods*, **Doron A. Peled**, Springer"

This is the end

- Hope you learned the fundamentals for specifying *your* Software Systems.
- Exam on 25.3.09.