

Model-Checking

CS511

Program Correctness

Model-Checking

Simple Verification Using Assertions

Turnstile Example

MEP

End States

Program Correctness

Main approaches to demonstrating that a program does what it's supposed to do:

1. Testing
2. Deductive verification
3. Model-checking

Testing

- ▶ Fast and simple way to detect errors
- ▶ Can never be **sure** there are no defects (cannot cover all the cases) – maybe tests weren't comprehensive enough

Testing shows the presence, not the absence of bugs¹

Testing Concurrent Programs

- ▶ More difficult since we would like to test all interleavings
- ▶ But since interleaving is controlled by OS scheduler, user cannot arrange arbitrary interleavings
- ▶ Consequence: **very** few of possible interleavings are tested

¹Dijkstra (1969) J.N. Buxton and B. Randell, eds, Software Engineering Techniques, April 1970, p. 16. Report on a conference sponsored by the NATO Science Committee, Rome, Italy, 27–31 October 1969.

Proving Programs Correct

- ▶ Holy Grail of computer science
- ▶ Using special **specification language**, describe
 1. State of program's variables
 2. How each programming language statement uses variables
- ▶ Specification language is mixture of mathematics & programming language

How to Prove a Program Correct

Hoare Triples

$$\llbracket A \rrbracket P \llbracket B \rrbracket$$

- ▶ P program
- ▶ A precondition
- ▶ B postcondition
- ▶ A and B are predicate logic formulae over an extended first-order language

Example

```
1  y:=1;  
2  z:=0;  
3  while (z!= x) {  
4      z:=z+1;  
5      y:=y*z  
6  }
```

Assertion

- ▶ Using hoare triples: $\llbracket \text{True} \rrbracket P \llbracket y = z! \wedge z = x \rrbracket$
- ▶ In prose: Under any state σ , if P terminates in a state ρ , then ρ satisfies $y = z! \wedge z = x$

Provable Assertion

- ▶ Prove $\llbracket \text{True} \rrbracket P \llbracket y = z! \wedge z = x \rrbracket$ in some deductive proof system

Sample Deductive Proof System for Partial Correctness

$$\frac{\llbracket A \rrbracket C_1 \llbracket B \rrbracket \quad \llbracket B \rrbracket C_2 \llbracket C \rrbracket}{\llbracket A \rrbracket C_1; C_2 \llbracket C \rrbracket} \text{ (COMPOSITION)}$$

$$\frac{}{\llbracket A\{x/E\} \rrbracket x := E \llbracket A \rrbracket} \text{ (ASSIGNMENT)}$$

$$\frac{\llbracket A \wedge B \rrbracket C_1 \llbracket D \rrbracket \quad \llbracket A \wedge \neg B \rrbracket C_2 \llbracket D \rrbracket}{\llbracket A \rrbracket \text{if } B \text{ then } \{C_1\} \text{ else } \{C_2\} \llbracket D \rrbracket} \text{ (CONDITIONAL)}$$

$$\frac{A \rightarrow A' \quad \llbracket A' \rrbracket P \llbracket B' \rrbracket \quad B' \rightarrow B}{\llbracket A \rrbracket P \llbracket B \rrbracket} \text{ (IMPLICATION)}$$

$$\frac{\llbracket A \wedge B \rrbracket C \llbracket A \rrbracket}{\llbracket A \rrbracket \text{while } B \{C\} \llbracket A \wedge \neg B \rrbracket} \text{ (WHILE-PARTIAL)}$$

Example of Partial Correctness Proof

$$\boxed{\llbracket T \rrbracket y := 1; z := 0; \text{while } (z! = x) \{ z := z + 1; y := y * z \} \llbracket y = z! \wedge z = x \rrbracket}$$

$$\frac{}{\llbracket y * (z + 1) = (z + 1)! \rrbracket z := z + 1 \llbracket y * z = z! \rrbracket}$$

$$\frac{}{\llbracket y = z! \wedge z \neq x \rrbracket z := z + 1 \llbracket y * z = z! \rrbracket}$$

$$\frac{}{\llbracket y * z = z! \rrbracket y := y * z \llbracket y = z! \rrbracket}$$

$$\frac{}{\llbracket y = z! \wedge z \neq x \rrbracket z := z + 1; y := y * z \llbracket y = z! \rrbracket}$$

$$\frac{}{\llbracket y = z! \rrbracket Q \llbracket y = z! \wedge z = x \rrbracket}$$

$$Q = \text{while } (z! = x) \{ z := z + 1; y := y * z \}.$$

$$\frac{}{\llbracket 1 = 0! \rrbracket y := 1 \llbracket y = 0! \rrbracket}$$

$$\frac{}{\llbracket y = 0! \rrbracket z := 0 \llbracket y = z! \rrbracket}$$

$$\frac{}{\llbracket T \rrbracket y := 1; z := 0; \llbracket y = z! \rrbracket}$$

$$\frac{}{\llbracket y = z! \rrbracket P \llbracket y = z! \wedge z = x \rrbracket}$$

$$\frac{}{\llbracket T \rrbracket y := 1; z := 0; P \llbracket y = z! \wedge z = x \rrbracket}$$

Drawbacks of Program Proof

- ▶ Proving that arbitrary program X has property Y is undecidable
- ▶ Precisely specifying all of program's intended actions is notoriously hard
 - ▶ Doing such a detailed spec & associated proofs usually much harder than writing & testing the program!
- ▶ Dynamic memory management (heap) is difficult to reason about
- ▶ Concurrency is even more difficult to reason about
 - ▶ See well-known books by Manna and Pnueli (1992,1995) or text by Apt et al (2009)

Research in Program Proof is Active

- ▶ Deductive verification is still too complicated for realistic programs/languages
- ▶ But it is growing fast!
 - ▶ The Atelier B system was used to develop part of the embedded software of the Paris metro line 14 and other railroad-related systems
 - ▶ Formally proved C compiler was developed using the Coq proof assistant (<http://compcert.inria.fr>)
 - ▶ Microsoft's hypervisor for highly secure virtualization was verified using VCC and the Z3 prover
 - ▶ L4-verified project developed a formally verified micro-kernel with high security guarantees, using analysis tools on top of the Isabelle/HOL proof assistant (<https://sel4.systems>)
 - ▶ <https://deepspec.org/main>

Research in Program Proof is Active

Many venues where research in program proof (and other program verification techniques) are published, including:

- ▶ International Conference on Computer Aided Verification (CAV)
- ▶ Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE)
- ▶ International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)
- ▶ Facebook's 2018 Testing and Verification Symposium (TAV)

Program Correctness

Model-Checking

Simple Verification Using Assertions

Turnstile Example

MEP

End States

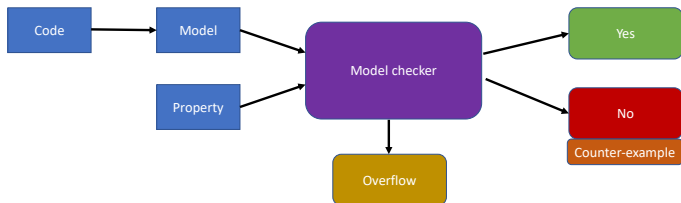
Model-Checking

1. Develop a **model** of the program

- ▶ This helps abstract away from unnecessary details
- ▶ Provides a different way of thinking about your problem
- ▶ Must be careful to not oversimplify

2. Prove properties of the model

- ▶ Model-checking involves using tools to analyze the model



Software Model Checking

Software model checking is the algorithmic analysis of programs to prove properties of their executions

- ▶ There is an extensive literature on this topic
- ▶ We only focus on one example (explicit state, automated, model-checking for temporal logic based on automata techniques)

- ▶ Survey:

Ranjit Jhala, Rupak Majumdar: Software model checking. ACM Comput. Surv. 41(4): 21:1-21:54 (2009)

Model-Checking

Two well-known **explicit-state** model-checkers for **distributed computing**

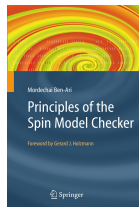
- ▶ Spin (we'll use this one)
 - ▶ Developed by Gerard Holzmann (1980s)
 - ▶ Awarded ACM's Software System Award in 2001
 - ▶ Example of use:
[Mars Code](#), Gerard J. Holzmann, Communications of the ACM, Vol. 57 No. 2, Pages 64-73, Feb 2014
- ▶ TLA+
 - ▶ Developed by Leslie Lamport (1994)
 - ▶ Example of use:
[How Amazon Web Services Uses Formal Methods](#), Chris Newcombe, Tim Rath, Fan Zhang, Bogdan Munteanu, Marc Brooker, Michael Deardeuff, Communications of the ACM, Vol. 58 No. 4, Pages 66-73, April 2015

Promela

- ▶ Spin uses Promela ((PRO)cess (ME)ta (LA)nguage) for representing models
- ▶ The aim of Promela is to model concurrent and distributed systems
- ▶ We'll look at some examples of Promela code
- ▶ They can be executed using spin

Tutorial:

- ▶ Principles of the Spin Model Checker, Mordechai Ben-Ari, Springer-Verlag London, 2008.



Promela Models

Consist of:

- ▶ type declarations
- ▶ channel declarations
- ▶ variable declarations
- ▶ process declarations
- ▶ `init` process

Corresponds to a (usually large, but) finite transition system, so

- ▶ no unbounded data
- ▶ no unbounded channels
- ▶ no unbounded processes
- ▶ no unbounded process creation

```
1  mtype = {MSG, ACK};
2  chan toS = ...
3  chan toR = ...
4  bool flag;
5
6  proctype Sender() {
7      ... process body ...
8  }
9
10 proctype Receiver() {
11     ...
12 }
13
14 init {
15     ...
16 }
```

Simple Sequential Program (eg1.pml)

```
1  active proctype P() {  
2      byte N = 10;  
3      byte sum = 0;  
4      byte i;  
5      for (i : 1 .. N) {  
6          sum = sum + i;  
7      }  
8      printf("The sum of the first %d numbers = %d\n",  
9          N, sum);  
10 }
```

- ▶ P is referred to as the process **type**
- ▶ **active** spawns a process type

Simple Sequential Program

```
1  active proctype P() {  
2      byte N = 10;  
3      byte sum = 0;  
4      byte i=1;  
5      do  
6          :: i > N -> break  
7          :: else ->  
8              sum = sum + i;  
9              i++  
10     od;  
11     printf("The sum of the first %d numbers = %d\n",  
12         N, sum);  
13 }
```

- ▶ Same as previous example only uses `do-od`

Simple Interleaving (eg2.pml)

```
1 byte n = 0;
2
3 active proctype P() {
4     n = 1;
5     printf("Process P, n = %d\n", n);
6 }
7
8 active proctype Q() {
9     n = 2;
10    printf("Process Q, n = %d\n", n);
11 }
```

Simple Interleaving with Race Condition (eg3.pm1)

```
1  byte      n = 0;
2
3  active proctype P() {
4      byte temp;
5      temp = n + 1;
6      n = temp;
7      printf("Process P, n = %d\n", n)
8  }
9
10 active proctype Q() {
11     byte temp;
12     temp = n + 1;
13     n = temp;
14     printf("Process Q, n = %d\n", n)
15 }
```

- ▶ Statements are atomic in Promela; interleaving occurs in an if- or do-statement (more later)

Simple Interleaving with Race Condition (eg4.pm1)

- ▶ Same as previous example but shorter
- ▶ Note the use of [2] and _pid (predefined variables start with an underscore)

```
1 byte    n = 0;
2
3 active [2] proctype P() {
4     byte temp;
5     temp = n + 1;
6     n = temp;
7     printf("Process P%d, n = %d\n", _pid, n);
8 }
```

Simple Interleaving with Race Condition (eg5.pm1)

- ▶ `init` is the first process that is activated
- ▶ `run` instantiates a process
- ▶ Convention: `run` expressions are enclosed in `atomic` so that all processes are instantiated before any of them begins execution

```
1  byte n;
2
3  proctype P(byte id; byte incr) {
4      byte temp;
5      temp = n + incr;
6      n = temp;
7      printf("Process P%d, n = %d\n", id, n)
8  }
9
10 init {
11     n = 1;
12     atomic {
13         run P(1, 10);
14         run P(2, 15)
15     }
16 }
```


Simple Interleaving with Race Condition (eg6.pm1)

- ▶ 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.
- ▶ 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

Simple Interleaving with Race Condition (eg6.pm1)

- ▶ (`_nr_pr == 1`) causes `init` to block until the expression is true (`_nr_pr` is number of processes currently running)

```
1  byte    n = 0;
2  proctype P() {
3      byte temp, i;
4      for (i:1..10) {
5          temp = n;
6          n=temp+1
7      }
8  }
9  init {
10     atomic {
11         run P();
12         run P()
13     }
14     (_nr_pr == 1);
15     printf("The value is %d\n", n);
16 }
```

Program Correctness

Model-Checking

Simple Verification Using Assertions

Turnstile Example

MEP

End States

Assert (eg8.pml)

```
1  byte    n = 0;
2  byte    finished = 0;
3
4  active [2] proctype P() {
5      byte i = 1;
6      byte temp;
7      for (i:1..10) {
8          temp = n;
9          n = temp + 1
10     }
11     finished++; /* Process terminates */
12 }
13
14 active proctype Finish() {
15     finished == 2; /* Wait for termination */
16     printf("n = %d\n", n);
17     assert (n > 2); /* Assert can't be 2 */
18 }
```

```
1  $ spin -a eg8.pml
2  $ gcc -o pan pan.c
3  $ ./pan
```

Verification in Spin using `assert`

- ▶ Spin reports:
 `pan:1: assertion violated (n>2) (at depth 90)`
- ▶ Spin also generates a trail counterexample
 - ▶ Stored in `eg8.pml.trail`
 - ▶ We can replay the counterexample with guided execution
 - ▶ We can also highlight the trace in the state diagram using SpinSpider

Closer Look at Output

```
1 pan:1: assertion violated (n>2) (at depth 88)
2 pan: wrote eg8.pml.trail
3
4 (Spin Version 6.4.8 -- 2 March 2018)
5 Warning: Search not completed
6   + Partial Order Reduction
7
8 Full statespace search for:
9   never claim           - (none specified)
10  assertion violations  +
11  acceptance   cycles  - (not selected)
12  invalid end states   +
13
14 State-vector 36 byte (size of a state), depth
15 reached 92 (longest path), errors: 1
16   138429 states (total number of states), stored
17   87813 states, matched
18   226242 transitions (= stored+matched)
19     0 atomic steps
20 hash conflicts:      4500 (resolved)
21
22 Stats on memory usage (in Megabytes):
23   8.449   equivalent memory usage for states (stored*(State-vector + overhead))
24   5.565   actual memory usage for states (compression: 65.86%)
25           state-vector as stored = 14 byte + 28 byte overhead
26   128.000 memory used for hash table (-w24)
27   0.534   memory used for DFS stack (-m10000)
28
29   134.003 total actual memory usage (memory used)
```

Inspecting the Trail from the Command Line

We could use `$spin -t eg8.pml`

```
1  n = 2
2  spin: eg8.pml:18, Error: assertion violated
3  spin: text of failed assertion: assert((n>2))
4  spin: trail ends after 89 steps
5  #processes: 3
6      TIMES = 10
7      n = 2
8      finished = 2
9  89:   proc  2 (Finish:1) eg8.pml:19 (state 4) <valid end state>
10  89:   proc  1 (P:1) eg8.pml:13 (state 12) <valid end state>
11  89:   proc  0 (P:1) eg8.pml:13 (state 12) <valid end state>
12  3 processes created
```

But this just shows the offending state, not the entire trail

Inspecting the Trail

Perhaps more informative is to use jSpin

1 P:1 1) else

0 P:1 1) else

1 P:1 1) temp = n

Process	Statement	P(1):temp
1	P:1 1) n = (temp+1)	0
0	P:1 1) temp = n	0

Pid, Process type, line number (??), statement, vars

Program Correctness

Model-Checking

Simple Verification Using Assertions

Turnstile Example

MEP

End States

Critical Section

```
1  bool wantP = false , wantQ = false;
2
3  active proctype P() {
4      do ::
5          printf("Non critical section P\n");
6          wantP = true;
7          printf("Critical section P\n");
8          wantP = false
9      od
10 }
11
12 active proctype Q() {
13     do ::
14         printf("Non critical section Q\n");
15         wantQ = true;
16         printf("Critical section Q\n");
17         wantQ = false
18     od
19 }
```

- ▶ Is mutual exclusion guaranteed? Use assertion
- ▶ Assertion requires knowing number of processes in their CSs

Critical Section

```
1  bool wantP = false, wantQ = false;
2  byte critical = 0;
3
4  active proctype P() {
5      do ::
6          printf("Non critical section P\n");
7          wantP = true;
8          critical++;
9          assert (critical == 1);
10         critical--;
11         printf("Critical section P\n");
12         wantP = false
13     od
14 }
15 active proctype Q() {
16     do ::
17         printf("Non critical section Q\n");
18         wantQ = true;
19         critical++;
20         assert (critical == 1);
21         critical--;
22         printf("Critical section Q\n");
```

Critical Section

- ▶ Result of verification:

pan:1: assertion violated (critical<=1) (at depth 20)

- ▶ Let's check the trail generated by spin
- ▶ Shorter counterexample produced by Random

```
1 Non critical section Q
2 1 Q:1 1) printf('Non cr
3 Non critical section P
4 0 P:1 1) printf('Non cr
5 1 Q:1 1) wantQ = 1
6 Process Statement          wantQ
7 0 P:1 1) wantP = 1         1
8 Process Statement          wantP          wantQ
9 1 Q:1 1) critical = (cr 1    1
10 Process Statement          critical        wantP          wantQ
11 0 P:1 1) critical = (cr 1          1          1
12 spin: cs.pml:25, Error: assertion violated
```

Revisiting Attempt III

```
1  global boolean wantP = false;
2  global boolean wantQ = false;

1  thread P: {
2      while (true) {
3          // non-critical section
4          wantP = true;
5          await !wantQ;
6          // CRITICAL SECTION
7          wantP = false;
8          // non-critical section
9      }
10 }

1  thread Q: {
2      while (true) {
3          // non-critical section
4          wantQ = true;
5          await !wantP;
6          // CRITICAL SECTION
7          wantQ = false;
8          // non-critical section
9      }
10 }
```

- ▶ Mutex: Yes
- ▶ Absence deadlock: No (we'll prove this using spin)
- ▶ Free from starvation: No

Attempt III in Promela

```
1  bool wantP = false, wantQ = false;
2  byte critical = 0;
3
4  active proctype P() {
5      do ::
6          printf("Non critical section P\n");
7          wantP = true;
8          wantQ == false;
9          critical++;
10         assert (critical <= 1);
11         critical--;
12         printf("Critical section P\n");
13         wantP = false
14     od
15 }
```

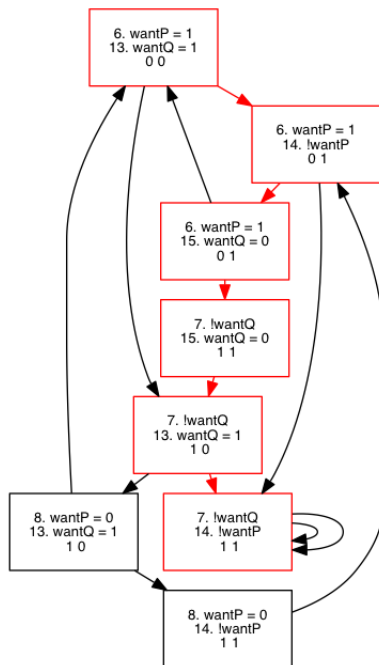
- ▶ We only list P, the full code is on the next slide
- ▶ Recall that an expression is **executable** iff it returns true
- ▶ The expr on line 9 blocks until it is true

Attempt III – Abbreviated

```
1  bool wantP = false, wantQ = false;
2  byte critical = 0;
3
4  active proctype P() {
5      do ::
6          wantP = true;
7          !wantQ;
8          critical++;
9          assert (critical <= 1);
10         critical--;
11         wantP = false
12     od
13 }
```

Verifying Attempt III

- Verification in spin reports
pan:1: invalid end state
(at depth 4)
- What is an invalid state? All
outgoing arrows point to same
state
- Here is the trail produced by
spin



Attempt III – Fix? – Atomic Entry Protocol

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4      do ::
5          atomic {
6              wantP = true;
7              !wantQ;
8          }
9              wantP = false
10     od
11 }
12
13 active proctype Q() {
14     do ::
15         atomic {
16             wantQ = true;
17             !wantP;
18         }
19             wantQ = false
20     od
21 }
```

Note: If any statement within the atomic sequence blocks, atomicity is lost, and other processes are then allowed to start executing statements. When the blocked statement becomes executable again, the execution of the atomic sequence can be resumed at any time, but not necessarily immediately.

Attempt III – Fix – Atomic Entry Protocol + Swap

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4      do ::
5          atomic {
6              !wantQ;
7              wantP = true;
8          }
9              wantP = false
10     od
11 }
12
13 active proctype Q() {
14     do ::
15         atomic {
16             !wantP;
17             wantQ = true;
18         }
19             wantQ = false
20     od
21 }
```

Attempt III with Fix – Verification

- ▶ Spin reports not errors:
●●● errors: 0 ●●●
- ▶ This means that there is not deadlock
- ▶ Exercise: Add assertions to check for mutual exclusion

Another possible fix

- ▶ Back off if there is contention
- ▶ This leads to our attempt IV (from the second set of slides of this course)

Revisiting Attempt IV

```
1  global boolean wantP = false;
2  global boolean wantQ = false;

1  thread P: {
2      while (true) {
3          // non-critical section
4          wantP = true;
5          while wantQ {
6              wantP = false;
7              wantP = true;
8          }
9          // CRITICAL SECTION
10         wantP = false;
11         // non-critical section
12     }
13 }

1  thread Q: {
2      while (true) {
3          // non-critical section
4          wantQ = true;
5          while wantP {
6              wantQ = false;
7              wantQ = true;
8          }
9          // CRITICAL SECTION
10         wantQ = false;
11         // non-critical section
12     }
13 }
```

- ▶ Mutex: Yes
- ▶ Absence deadlock: Yes
- ▶ Free from starvation: No

Revisiting Attempt IV

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4      do
5          :: wantP = true;
6          do
7              :: wantQ -> wantP = false; wantP = true
8              :: else -> break
9          od;
10         wantP = false
11     od
12 }
13
14 active proctype Q() {
15     do
16         :: wantQ = true;
17         do
18             :: wantP -> wantQ = false; wantQ = true
19             :: else -> break
20         od;
21     wantQ = false
22     od
```

Revisiting Attempt IV

- ▶ Check that there is no deadlock using spin
- ▶ Add assertions to check for mutual exclusion

Exercise: Write Dekker's Algorithm in Promela

```
1  global int turn = 1;
2  global boolean wantP = false;
3  global boolean wantQ = false;

1  thread P: {
2      while (true) {
3          // non-CS
4          wantP = true;
5          while wantQ
6              if (turn == 2) {
7                  wantP = false;
8                  await (turn==1);
9                  wantP = true;
10             }
11         // CS
12         turn = 2;
13         wantP = false;
14         // non-CS
15     }
16 }

1  thread Q: {
2      while (true) {
3          // non-CS
4          wantQ = true;
5          while wantP
6              if (turn == 1) {
7                  wantQ = false;
8                  await (turn==2);
9                  wantQ = true;
10             }
11         // CS
12         turn = 1;
13         wantQ = false;
14         // non-CS
15     }
16 }
```

Right to insist on entering is passed between the two processes

Exercise: Write Dekker's Algorithm in Promela

```
1  bool    wantp = false, wantq = false;
2  byte    turn = 1;
3
4  active proctype P() {
5      do
6          :: wantp = true;
7          do
8              :: !wantq -> break;
9              :: else ->
10                 if
11                     :: (turn == 1) /* no statements, leaves if */
12                     :: (turn == 2) ->
13                         wantp = false;
14                         (turn == 1);
15                         wantp = true
16                 fi
17             od;
18             wantp = false;
19             turn = 2
20         od
21 }
```


Program Correctness

Model-Checking

Simple Verification Using Assertions

Turnstile Example

MEP

End States

Additional Comment on End States

```
1  byte request = 0;
2
3  active proctype Server1() {
4      do
5          :: request == 1 ->
6              printf("Service 1\n");
7              request = 0;
8      od
9  }
10 active proctype Server2() {
11     do
12         :: request == 2 ->
13             printf("Service 2\n");
14             request = 0;
15     od
16 }
17 active proctype Client() {
18     request = 1;
19     request == 0;
20     request = 2;
21     request == 0;
22 }
```

Additional Comments on End States

- ▶ A process that does not terminate in its last instruction is said to be in an **invalid end state**
- ▶ Servers are always blocked at the guard of the do-statement waiting for it to become executable
- ▶ To avoid this: use a label to indicate that a control point is a valid end point, even if it is not the last instruction

```
1 active proctype Server1() {  
2     endserver:  
3     do  
4         :: request == 1 -> ...  
5     od  
6 }
```

Appendix

Installing Spin and jSpin

More Details on Promela Syntax

Installing Spin

- ▶ Binaries: <http://spinroot.com/spin/Bin/index.html>
- ▶ Uncompress and make executable (`chmod +x spin645_mac`)

Installing jSpin

- ▶ Installing jSpin

- ▶ Download:

- <http://www.weizmann.ac.il/sci-tea/benari/software-and-learning-materials/jspin>

- ▶ Compile and create .jar file

- ▶ Configuration:

- ▶ Create a jspin-5-0/bin directory

- ▶ Add binary file for spin in jspin-5-0/bin (eg. spin645_mac).

- ▶ Modify the following items in config.cfg:

- ```
SPIN=../bin/spin645_mac
```

- ```
C_COMPILER=/usr/bin/gcc
```

- ```
DOT=/usr/local/bin/dot
```

- ▶ Somewhat outdated reference manual: <http://wwinf.u-szeged.hu/~gombas/HSRV/jspin-user.pdf>

# Emacs and Dot

## ▶ Emacs

### ▶ Promela mode:

<https://github.com/rudi/promela-mode>

### ▶ Place in ~/.emacs.d/plugins

### ▶ Install by adding this to .emacs

```
(add-to-list 'load-path "~/.emacs.d/plugins")
(require 'promela-mode)
```

## ▶ Install dot

### ▶ brew install graphviz

# Execution using Spin/jSpin

- ▶ From command line: `../bin/spin645_mac count.pml`
- ▶ Using jSpin
- ▶ Modes:
  - ▶ Random
  - ▶ Interactive
  - ▶ Guided: follows the error trail that was produced by an earlier verification (not presented yet) run



# Bibliography

- ▶ Principles of the Spin Model Checker, Mordechai Ben-Ari, Springer, 2008 (reprinted April 11, 2013).
- ▶ <http://spinroot.com/spin/Doc/SpinTutorial.pdf>

# Appendix

Installing Spin and jSpin

More Details on Promela Syntax

# Promela Summary

| FEATURE   | C                      | PROMELA                  |
|-----------|------------------------|--------------------------|
| integers  | char, short, int, long | byte, short, int         |
| bit field | unsigned               | unsigned                 |
| floats    | float, double          | NONE                     |
| boolean   | int                    | bool                     |
| strings   | char, char*            | NONE                     |
| arrays    | yes                    | 1D & limited             |
| operators | many                   | mostly same              |
| if        | as usual               | similar to Erlang        |
| loops     | while, for, do         | do, similar to if        |
| output    | printf                 | printf                   |
| input     | scanf                  | NONE                     |
| functions | yes                    | NO                       |
| pointers  | yes                    | NO                       |
| enum      | enum                   | mtype                    |
| comments  | /* */ and //           | /* */                    |
| cpp       | full                   | 1-line #define, #include |

# If Syntax

- ▶ “Guarded commands” wrapped inside “if ... fi”

```
1 disc = b*b - 4*a*c;
2 if
3 :: disc > 0 ->
4 printf("two real roots\n")
5 :: disc < 0 ->
6 printf("no real roots\n")
7 :: disc == 0 ->
8 printf("duplicate real roots\n")
9 fi
```

# If Semantics

- ▶ First: evaluate all guards
- ▶ Then:
  - ▶ If no guard true: statement blocks until at least one guard becomes true (which could happen due to action of some concurrent process)
  - ▶ If one guard true: execute its command(s)
  - ▶ If more than one guard true: execute command(s) of randomly chosen guard

# Else

- ▶ Guard consisting of “else” keyword is true if all other guards are blocked
- ▶ Example:

```
1 disc = b*b - 4*a*c;
2 if
3 :: else ->
4 printf("two real roots\n")
5 :: disc < 0 ->
6 printf("no real roots\n")
7 :: disc == 0 ->
8 printf("duplicate real roots\n")
9 fi
```

# Do Syntax

- ▶ Similar to if statement
- ▶ Example: compute GCD by repeated subtraction

```
1 /* assume x and y are initialized */
2 int a = x, b = y;
3 do :: a > b -> a = a - b
4 :: b > a -> b = b - a
5 :: a == b -> break
6 od
7 printf("GCD(%d, %d) = %d\n", x, y, a);
```

- ▶ Notes:
  - No loop test; only way out is via break
  - Body consists of guarded commands
  - Some true guard is chosen at random
  - Block if no true guard

# Do Semantics

- ▶ Promela has no other type of loop
- ▶ Most common loop has only 2 guarded commands:

```
1 do
2 :: [exit test] -> break
3 :: else -> [body statements]
4 od
```

- ▶ This structure provides deterministic operation like:

```
1 while (not [exit test])
2 { [body statements] }
```



## Another Example

```
1 proctype P() {
2 int x = 15, y = 20;
3 int a = x, b = y;
4
5 do
6 :: a > b -> a = a - b
7 :: b > a -> b = b - a
8 :: a == b -> break
9 od
10 printf("GCD(%d, %d) = %d\n", x, y, a);
11 }
```

Note:

- ▶ `proctype P()` declares no-argument program P
- ▶ Can include arguments:

```
1 proctype P(int x, int y) {
2 int a = x, b = y;
3 etc. }
```

# Spawning a Process

- ▶ Can start processes using `run` operator: `run P(15, 20)`
- ▶ Also, can declare process with `active proctype`
  - ▶ Adding “active” means “define and run this program”
- ▶ To start two processes executing same code, use:  
`active [2] proctype P(int x, int y)`
- ▶ Can create an initial process that runs before any of the “proctype” processes
  - ▶ This process must be named `init`

# Predefined Variables

- ▶ `_pid` is process ID
- ▶ `_nr_pr` is number of active processes
- ▶ Examples:

```
1 printf("process %d: n goes from %d to %d\n", _pid, temp, n)
2
3 if
4 :: _nr_pr == 1 -> printf("at end n = %d\n", n);
5 fi
```

# Blocking Statements, I

- ▶ Concurrent programs must often wait for some event
- ▶ Possible to guard any statement
- ▶ This:

```
1 _nr_pr == 1 -> printf("at end n = %d\n", n)
```

is the same as:

```
1 if
2 :: _nr_pr == 1 -> printf("at end n = %d\n", n)
3 fi
```

## Blocking Statements, II

- ▶ “->” arrow is just syntactic sugar
- ▶ Can write expression by itself; if it doesn't evaluate to non-zero then program will block
- ▶ This:

```
1 _nr_pr == 1;
2 printf("at end n = %d\n", n)
```

is the same as:

```
1 _nr_pr == 1 -> printf("at end n = %d\n", n)
```

# Atomicity, I

- ▶ Individual Promela statements are atomic
- ▶ Warning! In Promela, expressions are statements too (hence expressions are atomic)
- ▶ Example – here, division by zero is possible:

```
1 if
2 :: a != 0 -> c = b / a ;
3 :: else -> c = b
4 fi
```

- ▶ In an if (and do) statement, interleaving may occur between the evaluation of the guard and the execution of the statement after the guard

## Atomicity, II

- ▶ It is tempting to regard the entirety of  $a \neq 0 \rightarrow c = b / a$  as atomic
- ▶ But it consists of two atomic parts,  $a \neq 0$  and  $c = b / a$
- ▶ Remember that this:

```
1 a != 0 -> c = b / a
```

could be written as:

```
1 a != 0; /* may block */
2 c = b / a
```

- ▶ The latter more obviously contains two atomic parts

# Atomicity, III

- ▶ To group statements together atomically use `atomic`

```
1 atomic {
2 a != 0; /* may block */
3 c = b / a
4 }
```

- ▶ If any statement within the atomic sequence blocks, atomicity is lost, and other processes may start executing statements.
- ▶ When the blocked statement becomes executable again, the execution of the atomic sequence can be resumed at any time (but it has to compete with other active processes)



# Atomic & Run

- ▶ `run` only starts a concurrent process
- ▶ `atomic` prevents execution of any other actions besides those in its body
- ▶ Therefore, to start a group of processes that should run concurrently:

```
1 atomic {
2 run P1 (...);
3 run P2 (...);
4 ...
5 run PN (...)
6 }
```

- ▶ At conclusion of `atomic` block: all processes have been started but none is yet running

# Variable Size

- ▶ Use smallest integer variable that will fit the need
- ▶ E.g., for integers known to be small use “byte” (8 bits) instead of “int” (32 bits)
- ▶ Reason: “verification” simulates all possible values of variable