Model-Checking

CS511

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 bugs[1]

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

[1] Dijkstra (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

```
y:=1;
z:=0;
while (z!= x) {
   z:=z+1;
   y:=y*z
}
```

Assertion

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

Provable Assertion

- Prove $[\![\mathit{True}]\!] P [\![y = z! \wedge z = x]\!]$ in some deductive proof system

Sample Deductive Proof System for Partial Correctness

$$\frac{[\![A]\!]\,C_1\,[\![B]\!] \quad [\![B]\!]\,C_2\,[\![C]\!]}{[\![A]\!]\,C_1;\,C_2\,[\![C]\!]}\ (\textsc{Composition})$$

$$\frac{}{[\![A\{x/E\}]\!]\,x := E\,[\![A]\!]}\ (\textsc{Assignment})$$

$$\frac{[\![A \wedge B]\!]\,C_1\,[\![D]\!] \quad [\![A \wedge \neg B]\!]\,C_2\,[\![D]\!]}{[\![A]\!]\,\texttt{if}\ B\ \texttt{then}\,\{C_1\}\,\texttt{else}\,\{C_2\}\,[\![D]\!]}\ (\textsc{Conditional})$$

$$\frac{A \to A' \quad [\![A']\!]\,P\,[\![B']\!] \quad B' \to B}{[\![A]\!]\,P\,[\![B]\!]}\ (\textsc{Implication})$$

$$\frac{[\![A \wedge B]\!]\,C\,[\![A]\!]}{[\![A]\!]\,\texttt{while}\ B\,\{C\}\,[\![A \wedge \neg B]\!]}\ (\textsc{While-Partial})$$

Example of Partial Correctness Proof

$$\boxed{[\![T]\!]\ y := 1; z := 0; \texttt{while}\,(z! = x)\,\{z := z + 1; y := y * z\}\ [\![y = z! \wedge z = x]\!]}$$

$$\dfrac{\dfrac{\dfrac{\dfrac{}{[\![y * (z + 1) = (z + 1)!]\!]\ z := z + 1\ [\![y * z = z!]\!]}}{[\![y = z! \wedge z \neq x]\!]\ z := z + 1\ [\![y * z = z!]\!]} \qquad \dfrac{}{[\![y * z = z!]\!]\ y := y * z\ [\![y = z!]\!]}}{[\![y = z! \wedge z \neq x]\!]\ z := z + 1; y := y * z\ [\![y = z!]\!]}}{[\![y = z!]\!]\ Q\ [\![y = z! \wedge z = x]\!]}$$

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

$$\dfrac{\dfrac{\dfrac{}{[\![1 = 0!]\!]\ y := 1\ [\![y = 0!]\!]} \qquad \dfrac{}{[\![y = 0!]\!]\ z := 0\ [\![y = z!]\!]}}{[\![T]\!]\ y := 1; z := 0;\ [\![y = z!]\!]} \qquad \dfrac{}{[\![y = z!]\!]\ Q\ [\![y = z! \wedge z = x]\!]}}{[\![T]\!]\ y := 1; z := 0; Q\ [\![y = z! \wedge z = x]\!]}$$

Drawbacks of Program Proof

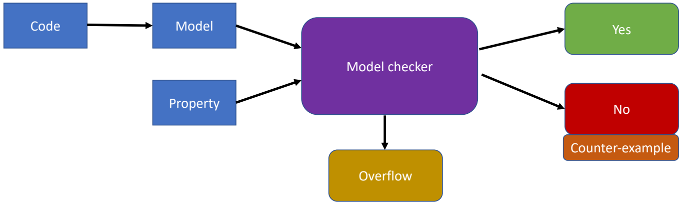
- Proving that arbitrary program X has property Y is undecidable
- Precisely specifying all of program's intended actions is notoriously hard
  - Doing such a detailed spec & associated proofs usually much harder than writing & testing the program!
- Dynamic memory management (heap) is difficult to reason 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

- 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`
  - Facebook's Infer tool based on Separation Logic `https://fbinfer.com`

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
   - Use 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 concurrent/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

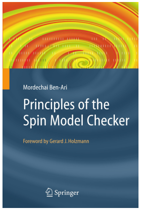
Model-Checking: Plan

1. Introduction to Promela
2. Assertion based model checking
3. Non-progress: livelock, starvation and fairness
4. LTL based model checking

Bibliography - Spin and Promela

Tutorial on Promela and Spin:

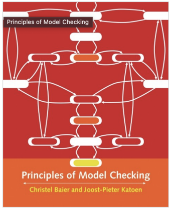
- Model Checking Concurrent Programs, Ian Barland, Moshe Vardi and John Greiner. Available here: https://cnx.org (search for title above) or download it here https://cnx.org/exports/cd5745fd-3270-46ee-9b64-6e4843b67c43@3.4.pdf/model-checking-concurrent-programs-3.4.pdf
- Principles of the Spin Model Checker, Mordechai Ben-Ari, Springer-Verlag London, 2008.


Mordechai Ben-Ari
Principles of the Spin Model Checker
Foreword by Gerard J. Holzmann
Springer

- Some slides: http://spinroot.com/spin/Doc/SpinTutorial.pdf

Bibliography - Foundations of Model Checking

- Principles of Model Checking, Christel Baier and Joost-Pieter Katoen, MIT Press, April 2008


Principles of Model Checking
Principles of Model Checking
Christel Baier and Joost-Pieter Katoen

Promela

- Spin uses Promela (PROcess MEta LAnguage) 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

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

```
mtype = {MSG, ACK};
chan toS = ...
chan toR = ...
bool flag;

proctype Sender() {
    ... process body ...
}

proctype Receiver() {
    ...
}

init {
    ...
}
```

Simple Sequential Program (eg1.pml)

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

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

Simple Sequential Program

```
active proctype P() {
  byte N = 10;
  byte sum = 0;
  byte i=1;
  do
  :: i > N -> break
  :: else ->
          sum = sum + i;
          i++
  od;
  printf("The sum of the first %d numbers = %d\n",
     N, sum);
}
```

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

Simple Interleaving (eg2.pml)

```
byte n = 0;

active proctype P() {
    n = 1;
    printf("Process P, n = %d\n", n);
}

active proctype Q() {
    n = 2;
    printf("Process Q, n = %d\n", n);
}
```

Simple Interleaving with Race Condition (eg3.pml)

```
byte    n = 0;

active proctype P() {
    byte temp;
    temp = n + 1;
    n = temp;
    printf("Process P, n = %d\n", n)
}

active proctype Q() {
    byte temp;
    temp = n + 1;
    n = temp;
    printf("Process Q, n = %d\n", n)
}
```

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

Simple Interleaving with Race Condition (eg4.pml)

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

```
byte     n = 0;

active [2] proctype P() {
    byte temp;
    temp = n + 1;
    n = temp;
    printf("Process P%d, n = %d\n", _pid, n);
}
```

Simple Interleaving with Race Condition (`eg5.pml`)

- `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

```
byte n;

proctype P(byte id; byte incr) {
  byte temp;
  temp = n + incr;
  n = temp;
  printf("Process P%d, n = %d\n", id, n)
}

init {
  n = 1;
  atomic {
    run P(1, 10);
    run P(2, 15)
  }
}
```

Simple Interleaving with Race Condition (`eg6.pml`)

- 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.pml)

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

```
byte    n = 0;
proctype P() {
    byte temp, i;
    for (i:1..10) {
      temp = n;
      n=temp+1
  }
}
init {
    atomic {
        run P();
        run P()
    }
    (_nr_pr == 1);
    printf("The value is %d\n", n);
}
```

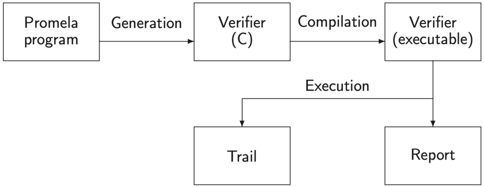
Assert (`eg8.pml`)

```
byte    n = 0;
byte    finished = 0;

active [2] proctype P() {
  byte  i = 1;
  byte  temp;
  for (i:1..10) {
    temp = n;
    n = temp + 1
  }
  finished++;  /* Process terminates */
}

active proctype Finish() {
  finished == 2;  /* Wait for termination */
  printf("n = %d\n", n);
  assert (n > 2); /* Assert can't be 2 */
}
```

Verification in Spin using Assertions



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

Verification in Spin using Assertions

- Since there was an assertion violation, it also generates a trail counterexample (`eg8.pml.trail`)

```
pan:1: assertion violated (n>2) (at depth 88)
pan: wrote eg8.pml.trail

(Spin Version 6.4.8 -- 2 March 2018)
Warning: Search not completed
    + Partial Order Reduction

Full statespace search for:
    never claim             - (none specified)
    assertion violations    +
    acceptance   cycles     - (not selected)
    invalid end states   +

State-vector 36 byte (size of a state), depth
reached 92 (longest path), errors: 1
   138429 states (total number of states), stored
    87813 states, matched
   226242 transitions (= stored+matched)
        0 atomic steps
hash conflicts:      4500 (resolved)

Stats on memory usage (in Megabytes):
    8.449   equivalent memory usage for states (stored*(State-vector + overhead))
    5.565   actual memory usage for states (compression: 65.86%)
            state-vector as stored = 14 byte + 28 byte overhead
  128.000   memory used for hash table (-w24)
    0.534   memory used for DFS stack (-m10000)
  134.003 total actual memory usage (memory used)
```

Inspecting the Trail

We can replay the counterexample in `eg8.pml.trail`

```
$ ./pan -r
```

```
  1:     proc  1 (P) eg8.pml:7 (state 1) [i = 1]
  2:     proc  1 (P) eg8.pml:7 (state 8) [((i<=10))]
  3:     proc  0 (P) eg8.pml:7 (state 1) [i = 1]
  4:     proc  0 (P) eg8.pml:7 (state 8) [((i<=10))]
  5:     proc  1 (P) eg8.pml:8 (state 3) [temp = n]
  6:     proc  0 (P) eg8.pml:8 (state 3) [temp = n]
  7:     proc  1 (P) eg8.pml:9 (state 4) [n = (temp+1)]
  8:     proc  1 (P) eg8.pml:7 (state 5) [i = (i+1)]
  9:     proc  1 (P) eg8.pml:7 (state 8) [((i<=10))]
 10:     proc  1 (P) eg8.pml:8 (state 3) [temp = n]
 ...
```

Critical Section

```
bool wantP = false, wantQ = false;

active proctype P() {
    do ::
        printf("Non critical section P\n");
        wantP = true;
        printf("Critical section P\n");
        wantP = false
    od
}

active proctype Q() {
    do ::
        printf("Non critical section Q\n");
        wantQ = true;
        printf("Critical section Q\n");
        wantQ = false
    od
}
```

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

Critical Section

```
bool wantP = false, wantQ = false;
byte critical = 0;

active proctype P() {
  do ::
       printf("Non critical section P\n");
       wantP = true;
       critical++;
       assert (critical == 1);
       critical--;
       printf("Critical section P\n");
       wantP = false
   od
}
active proctype Q() {
  do ::
       printf("Non critical section Q\n");
       wantQ = true;
       critical++;
       assert (critical == 1);
       critical--;
       printf("Critical section Q\n");
```

Critical Section

- We verify:

```
$ spin -a eg9.pml
$ gcc -o pan pan.c
$ ./pan
```

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

- Let's check the trail generated by spin (`pan -r`):

```
Non critical section Q
1 Q:1   1)  printf('Non cr
Non critical section P
0 P:1   1)  printf('Non cr
1 Q:1   1)  wantQ = 1
Process Statement           wantQ
0 P:1   1)  wantP = 1       1
Process Statement           wantP       wantQ
1 Q:1   1)  critical = (cr 1           1
Process Statement           critical    wantP       wantQ
0 P:1   1)  critical = (cr 1           1           1
spin: cs.pml:25, Error: assertion violated
```

Non-Progress Cycles

- SPIN can check for some simple liveness properties without the need to use Linear Temporal Logic
- We designate states as progress states by using a label prefixed with `progress`
- An infinite computation that does not include infinitely many occurrences of a progress state is called a *non-progress cycle*

Revisiting Attempt III

```
boolean wantP = false;
boolean wantQ = false;
```

```
Thread.start { //P
  while (true) {
   // non-critical section
   wantP = true;
   await (!wantQ);
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start { // Q
  while (true) {
   // non-critical section
   wantQ = true;
   await (!wantP);
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

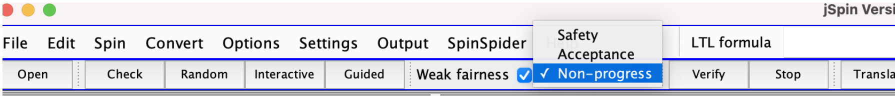
- Mutex: Yes
- Absence livelock: No (we'll prove this using spin)
- Free from starvation: No

Attempt III in Promela

```
bool wantP = false; bool wantQ = false;

active proctype P() {
    do ::
        wantP = true;
        do
        :: wantQ==false -> break
        :: else
        od;
progress1:
        wantP = false
    od
}
active proctype Q() {
    do ::
        wantQ = true;
        do
        :: wantP==false -> break
        :: else
        od;
progress2:
        wantQ = false
```

We Verify...



```
pan:1:  non-progress cycle (at depth 4) pan:  wrote
a4.pml.trail
```

We Verify...

The trail

```
starting claim 2
spin: couldn't find claim 2 (ignored)
1 Q:1   1)  wantQ = 1
Process Statement          wantQ
0 P:1   1)  wantP = 1      1
<<<<<START OF CYCLE>>>>>
Process Statement          wantP      wantQ
1 Q:1   1)  else           1          1
0 P:1   1)  else           1          1
spin: trail ends after 8 steps
#processes: 2
  8: proc  1 (Q:1) a4.pml:19 (state 5)
  8: proc  0 (P:1) a4.pml:7 (state 5)
2 processes created
Exit-Status 0
```

Revisiting Attempt IV

```
boolean wantP = false;
boolean wantQ = false;
```

```
Thread.start { //P
  while (true) {
   // non-critical section
   wantP = true;
   while wantQ {
     wantP = false;
     wantP = true;
   }
   // CRITICAL SECTION
   wantP = false;
   // non-critical section
  }
}
```

```
Thread.start {//Q
  while (true) {
   // non-critical section
   wantQ = true;
   while wantP {
     wantQ = false;
     wantQ = true;
   }
   // CRITICAL SECTION
   wantQ = false;
   // non-critical section
  }
}
```

- Mutex: Yes
- Absence of livelock: No
- Free from starvation: No

Revisiting Attempt IV

```
bool wantP = false, wantQ = false;

active proctype P() {
  do
  :: wantP = true;
     do
     :: wantQ -> wantP = false; wantP = true
     :: else  -> break
     od;
     wantP = false
    od
}

active proctype Q() {
   do
   :: wantQ = true;
      do
      :: wantP -> wantQ = false; wantQ = true
      :: else -> break
      od;
   wantQ = false
   od
```

Revisiting Attempt IV

- Check that there is no deadlock using spin
- Check that absence of livelock fails

Starvation

- Livelock: computation continues but no process makes actual progress
- Starvation: computation continues, and some processes make progress but others don't.

```
int turn = 1;
```

```
Thread.start { // P          Thread.start { // Q
  while (true) {              while (true) {
     await (turn==1);            await (turn==2);
     turn = 2;                   turn = 1;
  }                            }
}                            }
```

- Mutex: Yes
- Absence livelock: Yes
- Free from starvation: No (a process could remain indefinitely in its non-critical section)

Starvation

- Mutex: Yes
- Absence livelock: Yes
- Free from starvation: No (a process could remain indefinitely in its non-critical section)

Show that Attempt I does not enjoy freedom from starvation
Here is how you code a simple infinite loop:

```
do
:: else
od;
```

Exercise: Write Dekker's Algorithm in Promela

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

```
thread P: {
  while (true) {
   // non-CS
   wantP = true;
   while wantQ
     if (turn == 2) {
       wantP = false;
       await (turn==1);
       wantP = true;
     }
   // CS
   turn = 2;
   wantP = false;
   // non-CS
  }
}
```

```
thread Q: {
  while (true) {
   // non-CS
   wantQ = true;
   while wantP
     if (turn == 1) {
       wantQ = false;
       await (turn==2);
       wantQ = true;
     }
   // CS
   turn = 1;
   wantQ = false;
   // non-CS
  }
}
```

Right to insist on entering is passed between the two processes

Exercise: Write Dekker's Algorithm in Promela

```
bool     wantp = false, wantq = false;
byte     turn = 1;

active proctype P() {
    do
    ::  wantp = true;
        do
        :: !wantq -> break;
        :: else ->
           if
           :: (turn == 1)  /* no statements, leaves if */
           :: (turn == 2) ->
               wantp = false;
               (turn == 1);
               wantp = true
           fi
        od;
        wantp = false;
        turn = 2
    od
  }
```

Additional Comment on End States

```
byte request = 0;

active proctype Server1() {
  do
  :: request == 1 ->
          printf("Service 1\n");
          request = 0;
  od
}
active proctype Server2() {
  do
  :: request == 2 ->
          printf("Service 2\n");
          request = 0;
  od
}
active proctype Client() {
  request = 1;
  request == 0;
  request = 2;
  request == 0;
}
```

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

```
active proctype Server1() {
    endserver:
    do
    :: request == 1 -> ...
    od
}
```

Appendix

Installing Spin

- Binaries: `https://github.com/nimble-code/Spin`
- OS X: Make executable (`chmod +x spin651_mac64`)

Installing jSpin

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. `spin651_mac64`).
  - Modify the following items in config.cfg:

```
SPIN=../bin/spin651_mac64
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/spin651_mac64 count.pml`
- Using jSpin
- Modes:
  - Random
  - Interactive
  - Guided: follows the error trail that was produced by an earlier verification (not presented yet) run

Appendix

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`”

```
disc = b*b - 4*a*c;
if
:: disc > 0 ->
   printf("two real roots\n")
:: disc < 0 ->
   printf("no real roots\n")
:: disc == 0 ->
   printf("duplicate real roots\n")
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:

```
disc = b*b - 4*a*c;
if
:: else ->
   printf("two real roots\n")
:: disc < 0 ->
   printf("no real roots\n")
:: disc == 0 ->
   printf("duplicate real roots\n")
fi
```

Do Syntax

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

```
/* assume x and y are initialized */
int a = x, b = y;
do :: a > b -> a = a - b
   :: b > a -> b = b - a
   :: a == b -> break
od
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:

```
do
   :: [exit test] -> break
   :: else -> [body statements]
od
```

- This structure provides deterministic operation like:

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

Another Example

```
proctype P() {
    int x = 15, y = 20;
    int a = x, b = y;

    do
    :: a > b   ->   a = a - b
    :: b > a   ->   b = b - a
    :: a == b  ->   break
    od
    printf("GCD(%d, %d) = %d\n", x, y, a);
}
```

Note:

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

```
    proctype P(int x, int y) {
      int a = x, b = y;
      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:

```
printf("process %d: n goes from %d to %d\n", _pid, temp, n)

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

Blocking Statements, I

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

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

is the same as:

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

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

is the same as:

```
_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:

```
if
   :: a != 0 -> c = b / a ;
   :: else   -> c = b
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 != 0 -> c = b / a` as atomic
- But it consists of two atomic parts, `a != 0` and `c = b / a`
- Remember that this:

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

  could be written as:

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

- The latter more obviously contains two atomic parts

Atomicity, III

- To group statements together atomically use `atomic`

```
atomic {
    a != 0;      /* may block */
    c = b / a
}
```

- 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:

```
    atomic {
        run P1(...);
        run P2(...);
            ...
        run PN(...)
    }
```

- 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