Model-Checking

CS511

Program Correctness

Model-Checking

Simple Verification Using Assertions

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

- Test "lots of cases"
- More tests passed, more likely program is defect-free
- Can never be sure there are no defects – maybe tests weren't comprehensive enough
- Testing concurrent programs is extra difficult
  - We would like to test all interleavings
  - But since interleaving is controlled by OS scheduler, user cannot arrange arbitrary interleavings
- Also,

  Testing shows the presence, not the absence of bugs[1]

[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

$$[\![A]\!]\, P\, [\![B]\!]$$

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

Example

```
1 total := 0;
2 while (x>0)
3 {
4   x := x - 1;
5   total := total + x
6 }
```

$$[\![x = x_0 \wedge x \geq 0]\!]\, P\, [\![total = \sum_{i=0}^{x_0} i]\!]$$

- x_0 is a logical variable
- Logical variables do not appear in the program
- They only appear in pre and post conditions
- The state does not assign them a value

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

- Pre-condition: T
  - True: meaning no requirements are placed on initial values of variables
- Post-condition: $y = z! \wedge z = x$
- The Hoare triple reads:

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

Example of Partial Correctness Proof

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

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

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

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

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)

Summary of Program Correctness Techniques

- Testing cannot cover all the cases & the hardest cases to program are also hardest to test
  - Still of great use in practice
- Deductive verification is still too complicated for realistic programs/languages (but growing fast!)
  - Atelier B system: used to develop part of the embedded software of the Paris metro line 14
  - Formally proved C compiler was developed using the Coq proof assistant (COMPCert)
  - 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.
  - DeepSpec project headed be Appel (Princeton)

Program Correctness

Model-Checking

Simple Verification Using Assertions

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

Model-Checking

Two well-known explicit-state model-checkers

- Spin (we'll use this one)
  - Developed by Gerard Holzmann (1980s)
  - Awarded ACM's Software System Award in 2001
  - Example of use:

    Mars Code, By 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, By 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

Simple Sequential Program (eg1.pml)

```
1  active proctype P() {
2      byte N = 10;
3      byte sum = 0;
4
5      byte i;
6      i = 1;
7      do
8      :: i > N -> break
9      :: else ->
10             sum = sum + i;
11             i++
12     od;
13     printf("The sum of the first %d numbers = %d\n", N, sum);
14 }
```

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

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.pml)

```
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.pml`)

- 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.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

```
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.pml)

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

```
1  byte     n = 0;
2  proctype P() {
3      byte temp, i;
4      i = 1;
5      do
6      :: i > 10 -> break
7      :: else ->
8          temp = n;
9          n = temp + 1;
10         i++
11     od
12 }
13 init {
14     atomic {
15         run P();
16         run P()
17     }
18     (_nr_pr == 1);
19     printf("The value is %d\n", n);
```

Program Correctness

Model-Checking

Simple Verification Using Assertions

Assert (eg8.pml)

```
1  #define TIMES 10
2  byte    n = 0;
3  byte    finished = 0;
4
5  active [2] proctype P() {
6    byte  i = 1;
7    byte  temp;
8    do :: ( i > TIMES ) -> break
9       :: else ->
10         temp = n;
11         n = temp + 1;
12         i++
13   od;
14   finished++;  /* Process terminates */
15 }
16
17 active proctype Finish() {
18   finished == 2;  /* Wait for termination */
19   printf("n = %d\n", n);
20   assert (n > 2); /* Assert can't be 2 */
21 }
```

Verification in Spin using `assert`

- We next verify using spin
- Spin reports:

```
pan:1:  assertion violated (n>2) (at depth 90)
```

- Spin also generates a counterexample (a trail file)
- We can replay the counterexample with guided execution
- We can also highlight the trace in the state diagram using SpinSpider

Output Format

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

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

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
- Use a ghost variable `critical`

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");
23         wantQ = false
24   od
25 }
```

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

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

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

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

- 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 in Promela

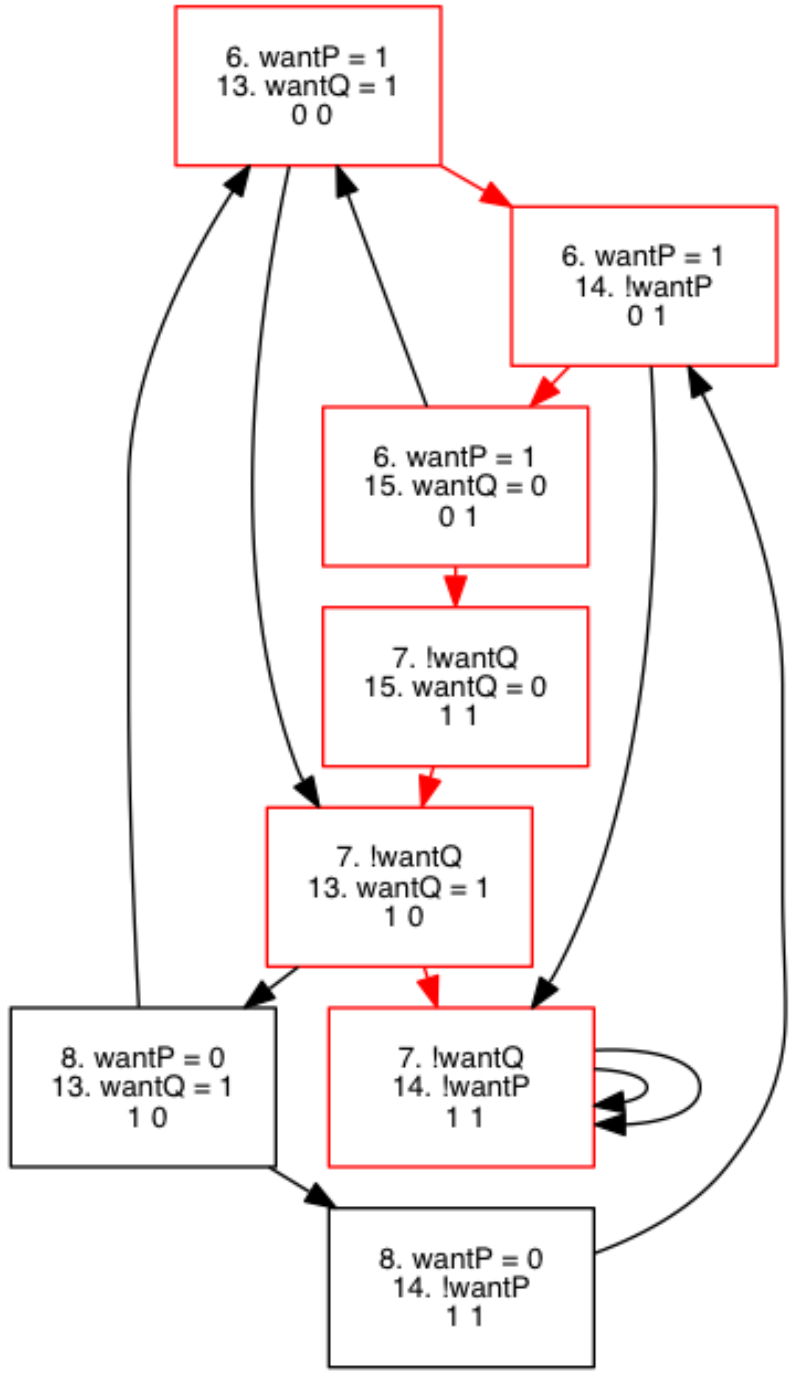
```
1  bool wantP = false, wantQ = false;
2  byte critical = 0;
3
4  active proctype P() {
5      do ::
6          printf("Non critical section P\n");
7          wantP = true;
8          wantQ == false;
9          critical++;
10         assert (critical <= 1);
11         critical--;
12         printf("Critical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

Attempt III – Abbreviated

```
1  bool wantP = false, wantQ = false;
2
3  active proctype P() {
4    do ::
5          wantP = true;
6          !wantQ;
7          critical++;
8          assert (critical <= 1);
9          critical--;
10         wantP = false
11   od
12 }
13
14 active proctype Q() {
15   do ::
16         wantQ = true;
17         !wantP;
18         critical++;
19         assert (critical <= 1);
20         critical--;
21         wantQ = false
22   od
23 }
```

Verifying Attempt III

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



Attempt III – Fix

- One easy fix is to have lines 5 and 6 below executed atomically
- The same for lines 13 and 14 below

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

Attempt III - Fix

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

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 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
23 }
```

Revisiting Attempt IV

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

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 }
```

Verification yields: `pan:1:  invalid end state (at depth 10)`

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

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).