

Software Engineering using Formal Methods

Modeling Concurrency

Wolfgang Ahrendt

12 September 2013

Concurrent Systems – The Big Picture

Concurrency: different processes trying not to run into each others' way

Main problem of concurrency: **sharing computational resources**

<http://www.youtube.com/watch?v=JgMB6nEv7K0>

<http://www.youtube.com/watch?v=G8eqmwUFi8>

shared resource = crossing, bikers = processes,
and a (data) race in progress, approaching a disaster.

Solutions to this must be carefully **designed** and **verified**, otherwise. . .

Concurrent Systems – The Big Picture



Focus of this Lecture

aim of SPIN-style model checking methodology:

exhibit design flaws in **concurrent** and **distributed** software systems

focus of this lecture:

- ▶ modeling and analyzing concurrent systems

focus of next lecture:

- ▶ modeling and analyzing distributed systems

Concurrent/Distributed systems difficult to get right

problems:

- ▶ hard to predict, **hard to form faithful intuition** about
- ▶ enormous combinatorial explosion of possible behavior
- ▶ interleaving prone to unsafe operations
- ▶ counter measures prone to deadlocks
- ▶ limited control—from within applications—over 'external' factors:
 - ▶ scheduling strategies
 - ▶ relative speed of components
 - ▶ performance of communication mediums
 - ▶ reliability of communication mediums

Testing Concurrent or Distributed System is Hard

We cannot exhaustively **test** concurrent/distributed systems

- ▶ lack of controllability
⇒ we miss failures in test phase
- ▶ lack of reproducibility
⇒ even if failures appear in test phase,
often impossible to analyze/debug defect
- ▶ lack of time
exhaustive testing exhausts the testers long before it exhausts
behavior of the system...

Mission of SPIN-style Model Checking

offer an efficient methodology to

- ▶ improve the design
- ▶ exhibit defects

of concurrent and distributed systems

Activities in SPIN-style Model Checking

1. model (critical aspects of) concurrent/distributed system with PROMELA
2. use assertions, temporal logic, ... to model crucial properties
3. use SPIN to check all possible runs of the model
4. analyze result, and possibly re-work 1. and 2.

The hardest part of Model Checking are 1. and 2.

Separate concerns of model vs. property! Check the property you want the model to have, not the one it happens to have.

Main Challenges of Modeling

expressiveness

model must be expressive enough to 'embrace' defects the real system could have

simplicity

model simple enough to be 'model checkable', theoretically and practically

Modeling Concurrent Systems in Promela

in the SPIN approach,
the cornerstone of modeling concurrent/distributed systems are

PROMELA processes

Initializing Processes

there is always an initial process prior to all others
often declared *implicitly* using 'active'

can be declared *explicitly* with key word 'init'

```
init {  
    printf("Hello□world\n")  
}
```

if *explicit*, `init` is used to start other processes with `run` statement

Starting Processes

processes can be started *explicitly* using `run`

```
proctype P() {  
    byte local;  
    ...  
}
```

```
init {  
    run P();  
    run P()  
}
```

each `run` operator starts copy of process (with copy of local variables)

`run P()` does *not* wait for P to finish

PROMELA's `run` corresponds to JAVA's **start**, *not* to JAVA's **run**

Atomic Start of Multiple Processes

by convention, run operators enclosed in atomic block

```
proctype P() {  
    byte local;  
    ...  
}
```

```
init {  
    atomic {  
        run P();  
        run P()  
    }  
}
```

effect: processes only start executing once all are created

(more on `atomic` later)

Joining Processes

following trick allows 'joining', i.e., waiting for all processes to finish

```
byte result;
```

```
proctype P() {
```

```
    ...
```

```
}
```

```
init {
```

```
    atomic {
```

```
        run P();
```

```
        run P();
```

```
    }
```

```
    (_nr_pr == 1); /*blocks until join*/
```

```
    printf("result_□=%d", result)
```

```
}
```

`_nr_pr` built-in variable holding number of running processes

`_nr_pr == 1` only 'this' process (init) is (still) running

Process Parameters

Processes may have formal parameters, instantiated by `run`:

```
proctype P(byte id; byte incr) {  
    ...  
}  
  
init {  
    run P(7, 10);  
    run P(8, 15)  
}
```

Active (Sets of) Processes

init can be made **implicit** by using the active modifier:

```
active proctype P() {  
    ...  
}
```

implicit init will run **one copy** of P

```
active [n] proctype P() {  
    ...  
}
```

implicit init will run **n copies** of P

Local and Global Data

Variables declared **outside** of the processes are **global** to all processes.

Variables declared **inside** a process are **local** to that processes.

```
byte n;
```

```
proctype P(byte id; byte incr) {  
    byte t;  
    ...  
}
```

n is **global**

t is **local**

Modeling with Global Data

pragmatics of modeling with global data:

shared memory of concurrent systems often modeled
by global variables of numeric (or array) type

status of shared resources (printer, traffic light, ...) often modeled
by global variables of Boolean or enumeration type
(`bool/mtype`).

communication mediums of distributed systems often modeled
by global variables of channel type (`chan`). (next lecture)

Interference on Global Data

```
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)  
}
```

how many outputs possible?

different processes can interfere on global data

Examples

1. `interleave0.pml`
SPIN simulation, SPINSPIDER automata + transition system
2. `interleave1.pml`
SPIN simulation, adding assertion, fine-grained execution model, model checking
3. `interleave5.pml`
SPIN simulation, SPIN model checking, trail inspection

limit the possibility of sequences being interrupted by other processes

weakly atomic sequence

can *only* be interrupted if a statement is not executable
defined in PROMELA by `atomic{ ... }`

strongly atomic sequence

cannot be interrupted at all
defined in PROMELA by `d_step{ ... }`

Deterministic Sequences

`d_step`:

- ▶ strongly atomic
- ▶ deterministic (like a single `step`)
- ▶ choices resolved in fixed way (always take the first option)
⇒ avoid choices in `d_step`
- ▶ it is an error if any statement within `d_step`,
other than the first one (called '*guard*'), blocks

```
d_step {  
    stmt1; ← guard  
    stmt2;  
    stmt3  
}
```

If `stmt1` blocks, `d_step` is **not entered**, and blocks as a whole.

It is an **error** if `stmt2` or `stmt3` block.

(Weakly) Atomic Sequences

atomic:

- ▶ weakly atomic
- ▶ can be non-deterministic

```
atomic {  
    stmt1; ← guard  
    stmt2;  
    stmt3  
}
```

If *guard* blocks, **atomic** is **not entered**, and blocks as a whole.

Once **atomic** is entered, control is kept until a statement blocks, and **only then** passed to another process.

Prohibit Interference by Atomicity

apply `atomic` or `d_step` to interference examples

Synchronization on Global Data

PROMELA has *no synchronization primitives*,
like semaphores, locks, or monitors.

instead, PROMELA inhibits concept of statement **executability**

executability addresses many issues in the interplay of processes

most known synchronization primitives (e.g. test & set, compare & swap, semaphores) can be modelled using executability and atomicity

Executability

Each statement has the notion of executability.

Executability of **basic statements**:

<i>statement type</i>	<i>executable</i>
assignment	always
assertion	always
print statement	always
<i>expression statement</i>	iff value not 0/false
send/receive statement	(next lecture)

Executability (Cont'd)

Executability of **compound statements**:

atomic resp. `d_step` statement is executable

iff

guard (i.e., the first inner statement) is executable

if resp. `do` statement is executable

iff

any of its alternatives is executable

an alternative is executable

iff

its guard (the first statement) is executable

(recall: in alternatives, “`->`” syntactic sugar for “`;`”)

Executability and Blocking

Definition (Blocking)

A **statement blocks** iff it is *not* executable.

A **process blocks** iff its location counter points to a blocking statement.

For each step of execution, the scheduler nondeterministically chooses a process to execute **among the non-blocking processes**.

Executability, resp. blocking are the key to PROMELA-style modeling of solutions to synchronization problems.
(to be discussed in the following)

The Critical Section Problem

archetypical problem of concurrent systems

given a number of looping processes, each containing a **critical section**

design an algorithm such that:

Mutual Exclusion At most one process is executing its critical section at any time.

Absence of Deadlock If *some* processes are trying to enter their critical sections, then *one* of them must eventually succeed.

Absence of (individual) Starvation If *any* process tries to enter its critical section, then *that* process must eventually succeed.

Critical Section Pattern

for demonstration, and simplicity:

(non)critical sections only `printf` statements

```
active proctype P() {
  do :: printf("P_non-critical_actions\n");
      /* begin critical section */
      printf("P_uses_shared_recourses\n");
      /* end critical section */
  od
}
```

```
active proctype Q() {
  do :: printf("Q_non-critical_actions\n");
      /* begin critical section */
      printf("Q_uses_shared_recourses\n");
      /* end critical section */
  od
}
```

No Mutual Exclusion Yet

need more infrastructure to achieve it:

adding two Boolean flags:

```
bool P_in_CS = false;
```

```
bool Q_in_CS = false;
```

```
active proctype P() {  
    do :: printf("P_in_critical_actions\n");  
        P_in_CS = true;  
        /* begin critical section */  
        printf("P_uses_shared_recourses\n");  
        /* end critical section */  
        P_in_CS = false  
    od  
}
```

```
active proctype Q() {  
    ...correspondingly...  
}
```

Show Mutual Exclusion VIOLATION with SPIN

adding assertions

```
bool P_in_CS = false;
bool Q_in_CS = false;

active proctype P() {
    do :: printf("P non-critical actions\n");
        P_in_CS = true;
        /* begin critical section */
        printf("P uses shared resources\n");
        assert(!Q_in_CS);
        /* end critical section */
        P_in_CS = false
    od
}

active proctype Q() {
    .....assert(!P_in_CS);.....
}
```


Mutual Exclusion by Busy Waiting

```
bool P_in_CS = false;
bool Q_in_CS = false;

active proctype P() {
    do :: printf("P_in_non-critical_actions\n");
        P_in_CS = true;
        do :: !Q_in_CS -> break
            :: else -> skip
        od;
        /* begin critical section */
        printf("P_in_uses_shared_recourses\n");
        assert(!Q_in_CS);
        /* end critical section */
        P_in_CS = false
    od
}

active proctype Q() { ...correspondingly... }
```

Mutual Exclusion by Blocking

instead of Busy Waiting, process should

- ▶ release control
- ▶ continuing to run only when exclusion properties are fulfilled

We can use **expression statement** `!Q_in_CS`,
to let process P **block** where it should not proceed!

Mutual Exclusion by Blocking

```
active proctype P() {
  do :: printf("P_non-critical_actions\n");
      P_in_CS = true;
      !Q_in_CS;
      /* begin critical section */
      printf("P_uses_shared_recourses\n");
      assert(!Q_in_CS);
      /* end critical section */
      P_in_CS = false
od
}
```



```
active proctype Q() {
  ...correspondingly...
}
```

Verify Mutual Exclusion of this

SPIN error (invalid end state)

⇒ deadlock

can make pan ignore the deadlock: `./pan -E`

SPIN still reports assertion violation(!)

Proving Mutual Exclusion

- ▶ mutual exclusion (ME) cannot be shown by SPIN
- ▶ P/Q_in_CS sufficient for *achieving* ME
- ▶ P/Q_in_CS *not* sufficient for *proving* ME

need more infrastructure:

ghost variables, only for proving / model checking

Show Mutual Exclusion with Ghost Variable

```
int critical = 0;

active proctype P() {
  do :: printf("P_\u25a1non-critical_\u25a1actions\n");
      P_in_CS = true;
      !Q_in_CS;
      /* begin critical section */
      critical++;
      printf("P_\u25a1uses_\u25a1shared_\u25a1recourses\n");
      assert(critical < 2);
      critical--;
      /* end critical section */
      P_in_CS = false
od
}

active proctype Q() {
  ...correspondingly...
}
```

Verify Mutual Exclusion of this

SPIN (./pan -E) shows no assertion is violated

⇒ mutual exclusion is verified

still SPIN (without -E) reports (invalid end state)

⇒ deadlock

Deadlock Hunting

Invalid End State:

- ▶ A process does not finish at its end
- ▶ OK if it is not crucial to continue – see last lecture
- ▶ Two or more inter-dependent processes do not finish at the end
Real **deadlock**

Find Deadlock with SPIN:

- ▶ Verify to produce a failing run trail
- ▶ Simulate to see how the processes get to the interlock
- ▶ Fix the code, not using the `end...: labels` or `-E switch` ;)

Atomicity against Deadlocks

solution:

checking and setting the flag in one atomic step

```
atomic {  
    !Q_in_CS;  
    P_in_CS = true  
}
```

Variations of Critical Section Problem

- ▶ designated artifacts for verification:
 - ▶ ghost variables (verification only)
 - ▶ temporal logic (later in the course)
- ▶ max n processes allowed in critical section modeling possibilities include:
 - ▶ counters instead of booleans
 - ▶ semaphores (see demo)
- ▶ more fine grained exclusion conditions, e.g.
 - ▶ several critical sections (Leidestraat in Amsterdam)
 - ▶ writers exclude each other and readers
readers exclude writers, but not other readers
 - ▶ FIFO queue semaphores, for fairly choosing processes to enter
- ▶ ... and many more

Why Not Critical Section in Single Atomic Block?

Actually possible in this case.

Also in interleaving example (counting via `temp`, see above).

But:

- ▶ does not carry over to variations (see previous slide)
- ▶ `atomic` only weakly atomic!
- ▶ `d_step` excludes any nondeterminism!

Using `atomic` and `d_step` too heavily, for too large blocks, can result in well-behaved models, while modelling the wrong system.