Formal Methods for Software Development
Modeling Concurrency

Wolfgang Ahrendt

05 September 2017
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=G8eqymwUFi8

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

Solutions to this must be carefully designed and verified, otherwise...
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
▶ 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
We cannot exhaustively test concurrent/distributed systems

- lack of controllability
  ⇒ we miss failures in test phase

- lack of reproducability
  ⇒ 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 \textit{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. state crucial properties with assertions, temporal logic, ...
3. use Spin to check all possible runs of the model
4. analyze result, possibly re-work 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 must be simple enough to be ‘model checkable’, theoretically and practically
In the Spin approach, the cornerstone of modeling concurrent/distributed systems are Promela processes.
Initializing Processes

Can be declared *implicitly* using ‘active’.

Can be declared *explicitly* with key word ‘init’

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

`init` is used to start other processes with `run` statement.
Processes can be started \textit{explicitly} using \texttt{run}

\begin{verbatim}
proctype P() {
    byte local;
    ...}

init {
    run P();
    run P();
    run P();
}
\end{verbatim}

Each \texttt{run} operator starts copy of process (with copy of local variables)
\texttt{run P()} does \textit{not} wait for \texttt{P} to finish

\texttt{(PROMELA's run corresponds to JAVA's \texttt{start}, not to JAVA's \texttt{run})}
Atomic Start of Multiple Processes

By convention, `run` operators enclosed in `atomic` block

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

```plaintext
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’: waiting for all processes to finish

```c
byte result;

proctype P() {
    ...
}

init {
    atomic {
        run P();
        run P()
    }
    (_nr_pr == 1); /*blocks until join*/
    printf("result =\n%d", result)
}
```

_nr_pr built-in variable holding number of running processes
_nr_pr == 1 only ‘this’ process (init) is (still) running
Processes may have formal parameters, instantiated by run:

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

init {
    run P(7, 10);
    run P(8, 15)
}
```
init can be made implicit by using the active modifier:

```plaintext
active proctype P() {
    ... 
}
Implicit init will run one copy of P
```

```plaintext
active [n] proctype P() {
    ... 
}
Implicit init will run n copies of P
```
Variables declared *outside* of the processes are *global* to all processes.

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

byte n;

```c
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)
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?
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
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 modeled using executability and atomicity.
Executability

Each statement has the notion of executability.

Executability of basic statements:

<table>
<thead>
<tr>
<th>statement type</th>
<th>executable</th>
</tr>
</thead>
<tbody>
<tr>
<td>assignment</td>
<td>always</td>
</tr>
<tr>
<td>assertion</td>
<td>always</td>
</tr>
<tr>
<td>print statement</td>
<td>always</td>
</tr>
<tr>
<td>expression statement</td>
<td>iff value not 0/false</td>
</tr>
<tr>
<td>send/receive statement</td>
<td>(next lecture)</td>
</tr>
</tbody>
</table>

Definition (Expression Statement)

An expression statement is a statement only consisting of an expression.
Executability (Cont’d)

Executability of compound statements:

if resp. do statement is executable
    iff
any of its alternatives\(^1\) is executable

An alternative is executable
    iff
its guard (the first statement) is executable
(Recall: in alternatives, “→” syntactic sugar for “;”)

(Inspect end.pml)

\(^1\text{alternative} = \text{list of statements}\)
**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.
Definition (Deadlock (simplified))

Let $CRP$ be the set of currently running processes.
A deadlock is a point in the execution where
- $CRP > 0$
- all $p \in CRP$ are blocking

(Verify end.pml)
Valid End States

Definition (End Location)

End locations of a process $P$ are:

- $P$’s textual end
- each location marked with an end label: “endxxx:”
Deadlock

Definition (Deadlock (full version))

Let $CRP$ be the set of currently running processes. Let $NEL \subseteq CRP$ be the set of (currently running) processes which are not at a valid end location.

A deadlock is a point in the execution where

- $NEL > 0$
- all $p \in NEL$ are blocking
Deadlock Detection

SPIN checks deadlocks per default!
⇒ No need to specify deadlock freedom.

Deadlock signaled by:

- ‘invalid end state’ error (in verification mode)
- ‘timeout’ in simulation mode

Deadlock check can be switched off by ./pan -E

(Fix end.pml)
Atomicity

limit the possibility of sequences being interrupted by other processes

**weakly atomic sequence**
- can only be interrupted if a statement blocks
- defined in Promela by `atomic{list_of_statements}`

**strongly atomic sequence**
- cannot be interrupted at all
- defined in Promela by `d_step{list_of_statements}`
atomic resp. d_step statement is executable
iff
guard (i.e., the first inner statement) is executable
Deterministic Sequences

d_step:

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

```plaintext
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 in this case passed to another process.
The Critical Section Problem

Archetypal problem of concurrent systems

**Critical section**: Section of code/model where interference of other processes can cause problems

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:
Noncritical and critical sections only printf statements here

active proctype P() {
    do :: printf("P\non-critical\actions\n");
        /* begin critical section */
        printf("P\uses\shared\resources\n")
        /* end critical section */
    od
}

active proctype Q() {
    do :: printf("Q\non-critical\actions\n");
        /* begin critical section */
        printf("Q\uses\shared\resources\n")
        /* end critical section */
    od
}
No Mutual Exclusion Yet

More infrastructure to achieve ME.
Adding two Boolean flags:

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

```c
active proctype P() {
    do :: printf("P\nnon-critical\nactions\n");
    P_in_CS = true;
    /* begin critical section */
    printf("P\nuses\nshared\nresources\n");
    /* end critical section */
    P_in_CS = false
    od
}
```

```c
active proctype Q() {
    ...correspondingly...
}
```
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); ..........
}
bool P_in_CS = false;
bool Q_in_CS = false;

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

active proctype Q() { ...correspondingly... }
Mutual Exclusion by Blocking

Instead of Busy Waiting, process should

1. yield control,
2. continue to run only when exclusion properties becomes true again.

We can use expression statement \texttt{!Q\_in\_CS},
to let process \texttt{P} \texttt{block} where it should not proceed!
Mutual Exclusion by Blocking

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

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

Verify Mutual Exclusion of this

Verify with $\text{SPIN}$

$\text{SPIN}$ error (invalid end state) ⇒ deadlock

can make $\text{pan}$ ignore the deadlock: $./\text{pan} -E$

$\text{SPIN}$ still reports assertion violation(!)
Proving Mutual Exclusion

In this example:

- mutual exclusion (ME) cannot be shown by \textit{Spin}
- \textit{P/Q\_in\_CS} sufficient for \textit{achieving} ME
- \textit{P/Q\_in\_CS} \textit{not} sufficient for \textit{proving} ME

Need more infrastructure.

\textbf{Ghost variables}: variables for verification, not for modeling
Show Mutual Exclusion with Ghost Variable

```c
int critical = 0;

active proctype P() {
    do :: printf("P␣non-critical␣actions\n");
    P_in_CS = true;
    !Q_in_CS;
    /* begin critical section */
    critical++;
    printf("P␣uses␣shared␣resources\n");
    assert(critical < 2);
    critical--;
    /* end critical section */
    P_in_CS = false
    od
}

active proctype Q() {
    ...correspondingly...
}
```
Verify Mutual Exclusion of this

\texttt{SPIN} (.\texttt{/pan} -E) shows no assertion is violated
⇒ mutual exclusion is verified

Still \texttt{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 – add end labels (see end.pml)
- If it is crucial to continue:
  Real deadlock

Address 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 nor -E option)
Atomicity against Deadlocks

solution:

checking and setting the flag in one atomic step

(demonstrate that in csGhost.pml)

```plaintext
atomic {
!Q_in_CS;
P_in_CS = true
}
```
Variations of Critical Section Problem

- Verification artifacts:
  - ghost variables (‘verification only’ variables)
  - 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?

- Does not carry over to variations (see previous slide).
- `atomic` only weakly atomic!
- `d_step` excludes any nondeterminism!
- Most important: this misses the point. We verify effectiveness of `atomic`, not of the modeled protection solution!

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