Applications of Formal Verification
Model Checking: Modeling Concurrency

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov | SS 2010
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
- (plus: staring with Temporal Logic Model Checking)
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 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 SPIN-style Model Checking

offer an efficient methodology to
- improve the design
- exhibit defects

of concurrent and distributed systems
Activities in \texttt{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.

I claim:
The hardest part of Model Checking is 1.
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
corner stone of modeling concurrent, and distributed, systems in SPIN approach are **PROMELA processes**
there is always an initial process prior to all others present *implicitly* when using ‘active’

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

```c
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

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

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

effect: processes only start executing once all are created
Joining Processes

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

```c
byte result;

proctype P() {
    ....
}

init {
    atomic {
        run P();
        run P();
    }
    (_nr_pr == 1) ->
        printf("result =%d", result)
}
```

_\_nr\_pr \ built in variable holding number of running processes
_\_nr\_pr = 1 \ only init is running (anymore)
Processes may have formal parameters, instantiated by `run`:

```plaintext
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 temp;
   ...
}

n is global

temp is local
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).
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 now?

different processes can interfere on global data
Examples

1. **interleave0.pml**
   SPIN simulation, SPINSPIIDER automata + transition system

2. **interleave1.pml**
   SPIN simulation, SPINSPIIDER automata + transition system

3. **interleave5.pml**
   SPIN simulation, SPIN model checking, trail inspection
Atomicity

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
  can not be interrupted at all
  defined in PROMELA by d_step{ ... }
Deterministic Sequences

d_step:

- strongly atomic
- deterministic
- nondeterminism resolved in fixed way
  ⇒ good style to avoid nondeterminism 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
apply \texttt{d\_step} to interference example
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
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>assignments</td>
<td>always</td>
</tr>
<tr>
<td>assertions</td>
<td>always</td>
</tr>
<tr>
<td>print statements</td>
<td>always</td>
</tr>
<tr>
<td>expression statements</td>
<td>iff value not 0/false</td>
</tr>
<tr>
<td>send/receive statements</td>
<td>(coming soon)</td>
</tr>
</tbody>
</table>
Executability (Cont’d)

Executability of compound statements:

atomic resp. d_step statement is executable
iff
guard (the first statement within) 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 “;”)

Prof. Dr. Bernhard Beckert · Dr. Vladimir Klebanov – Applications of Formal Verification  
SS 2010
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 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 it’s critical section 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

```c
active proctype P() {
    do :: printf("Noncritical section P\n");
    /* begin critical section */
    printf("Critical section P\n");
    /* end critical section */
    od
}

active proctype Q() {
    do :: printf("Noncritical section Q\n");
    /* begin critical section */
    printf("Critical section Q\n");
    /* end critical section */
    od
}
```
No Mutual Exclusion Yet

need more infrastructure to achieve it: adding two Boolean flags:

```c
bool inCriticalP = false;
bool inCriticalQ = false;
```

```c
active proctype P() {
    do :: printf("Non-critical section P\n");
    /* begin critical section */
    inCriticalP = true;
    printf("Critical section P\n");
    inCriticalP = false
    /* end critical section */
    od
}
```

```c
active proctype Q() {
    ...correspondingly...
}
```
Show Mutual Exclusion Violation with SPIN

adding assertions

```c
bool inCriticalP = false;
bool inCriticalQ = false;

active proctype P() {
    do ::
        printf("Non-critical section P\n");
        /* begin critical section */
        inCriticalP = true;
        printf("Critical section P\n");
        assert(!inCriticalQ);
        inCriticalP = false
        /* end critical section */
    od
}

active proctype Q() {
    .......assert(!inCriticalP); .......
}
```
bool inCriticalP = false;
bool inCriticalQ = false;

active proctype P() {
    do :: printf("Non-critical section P\n");
    /* begin critical section */
inCriticalP = true
    do :: !inCriticalQ -> break
         :: else -> skip
    od;
    printf("Critical section P\n");
    assert(!inCriticalQ);
inCriticalP = false
    /* end critical section */
    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** `!inCriticalQ`, to let process `P` **block** where it should not proceed!
bool inCriticalP = false;
bool inCriticalQ = false;

active proctype P() {
    do :: printf("Non-critical section P\n");
    /* begin critical section */
inCriticalP = true;
!inCriticalQ;
printf("Critical section P\n");
assert(!inCriticalQ);
inCriticalP = false
/* end critical section */
    od
}

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

\texttt{SPIN}
still errors (invalid end state)
⇒ deadlock
can make \texttt{pan} ignore the deadlock: \texttt{./pan -E}
\texttt{SPIN} then proves mutual exclusion
Deadlock Hunting

find Deadlock with SPIN
Atomicity against Deadlocks

solution:

checking and setting the flag in one atomic step

```java
atomic {
    !inCriticalQ;
    inCriticalP = true
}
```
Variations of Critical Section Problem

the example was simplistic indeed variations:

- use other means for verification:
  - ghost variables (verification only)
  - temporal logic (next lecture)

- 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 queues for entering sections (full semaphores)

- ... and many more
Solving CritSectPr with atomic/d_step only?

actually possible in this case (demo)
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!