The SPIN Model Checker

Metodi di Verifica del Software
Andrea Corradini
Lezione 1
2013

Slides liberamente adattate da “Logic Model Checking”,
per gentile concessione di Gerard J. Holzmann
http://spinroot.com/spin/Doc/course/
Why focus on SPIN?

- directly targets *software*, rather than hardware verification
- good example of the *automata theoretic* approach
- better to understand one system really well, so that you can use it effectively, rather than many different systems partially (?)
- based on well-understood theory of $\omega$-automata and linear temporal logic
- 2001 ACM Software Systems Award (other winning software systems include: Unix, TCP/IP, WWW, Tcl/Tk, Java)
- distributed freely as research tool, well-documented, actively maintained, growing user-base, users in both academia and industry
- annual Spin user workshops series held since 1995
types of correctness requirements

• some requirements are standard:
  – a system (e.g., an OS) should not be able to deadlock
  – no process should be able to starve another
  – no explicitly stated assertion inside a process should ever fail
• the most important requirements are application specific:
  – system invariants, process assertions
  – *effective progress* requirements
  – proper termination
  – general *causal* and *temporal* relations on states
    • e.g., when a request is issued eventually a reply is returned
  – fairness assumptions,
    • e.g., about process scheduling
  – etc. etc.
the choice of the model depends on the requirements that must be checked

- a good model is always an *abstraction* of reality
  - it should have *less detail* than the artifact being modeled
  - the level of detail is selected based on its relevance to the correctness requirements
  - the objective is to gain *analytical power* by reducing detail

- the purpose of a model is to *explain and predict*
  - if it can do neither because it is either too approximate or too detailed, it is *not* a good model

- a model is a *design aid*
  - it often goes through different versions, describing different aspects of reality, and can slowly become more *accurate*, *without* becoming more detailed

\[\text{accuracy} \neq \text{detail} \]
building verification models

- we want to be able to make separate statements about system *design* and about system *requirements*
- therefore we will need two notations/formalisms
  - one for specifying behavior (system design)
  - one for specifying requirements (correctness properties)
- the two types of statements combined define a *verification model*
- a model checker can now:
  - check that the behavior specification (the design) is logically consistent with the requirements specification (the desired properties of the design)
  - the formalism must be defined in such a way that we can guarantee the *decidability* of any property we can state for any system we can specify
Spin verification models are used to define *abstractions* of distributed system designs

- the specification language must support all essential aspects of distributed systems software, and discourage the specification of any redundant detail
- there are 3 basic types of objects in a Spin verification model:
  - asynchronous processes
  - global and local data objects
  - message channels
hello world as a “Spin model”

active proctype main()
{
    printf(“hello world\n”)
}

these are keywords
‘main’ is not a keyword
no semi-colon here…

this is a bit like C

a simulation run:

```bash
$ spin hello.pml
hello world
1 process created
$
```

a verification run:

```bash
$ spin -a hello.pml
$ gcc -o pan pan.c
$ ./pan
... depth reached 2, errors: 0
$
```
a more interesting example: two processes
a card reader and a line printer

process 1

- ?A
- ?B
- !A
- !B

process 2

- ?B
- ?A
- !B
- !A

?A reserve printer device
?B reserve card reader

!A release printer device
!B release card reader
the corresponding Spin model
(don’t worry about the details just yet)

```bash
$ cat generic.pml
bool printer = true; /* initially both devices */
bool reader = true; /* are available */

active [2] proctype user()
{
    do
        :: (printer) -> printer = false;
        :: (reader) -> reader = false;
        /* print cards */
        printer = true; /* available */
        reader = true
        :: (reader) -> reader = false;
        :: (printer) -> printer = false;
        /* print cards */
        printer = true;
        reader = true
    od
}

$ 
```
a simulation of 20 steps

```
$ spin -v -u20 generic.pml

0:    proc  - (:root:) creates proc  1 (user)
1:    proc  0 (user) line   6 "generic.pml" (state 13)[(printer)]
2:    proc  0 (user) line   7 "generic.pml" (state 2) [printer = 0]
3:    proc  0 (user) line   8 "generic.pml" (state 3) [(reader)]
4:    proc  1 (user) line   6 "generic.pml" (state 13)[(reader)]
5:    proc  0 (user) line   8 "generic.pml" (state 4) [reader = 0]
6:    proc  1 (user) line  13 "generic.pml" (state 8) [reader = 0]
7:    proc  0 (user) line  10 "generic.pml" (state 5) [printer = 1]
8:    proc  1 (user) line  14 "generic.pml" (state 9) [(printer)]
9:    proc  1 (user) line  14 "generic.pml" (state 10)[printer = 0]
10:   proc  0 (user) line  11 "generic.pml" (state 6) [reader = 1]
11:   proc  0 (user) line  19 "generic.pml" (state 14)[.(goto)]
12:   proc  0 (user) line   6 "generic.pml" (state 13)[(reader)]
13:   proc  1 (user) line  16 "generic.pml" (state 11)[reader = 1]
14:   proc  1 (user) line  17 "generic.pml" (state 12)[printer = 1]
15:   proc  1 (user) line  19 "generic.pml" (state 14)[.(goto)]
16:   proc  1 (user) line   6 "generic.pml" (state 13)[(reader)]
17:   proc  0 (user) line  13 "generic.pml" (state 8) [reader = 0]
18:   proc  1 (user) line  13 "generic.pml" (state 8) [reader = 0]
19:   proc  0 (user) line  14 "generic.pml" (state 9) [(printer)]
20:   proc  1 (user) line  14 "generic.pml" (state 9) [(printer)]

--------------
depth-limit (-u20 steps) reached
#processes: 2
    printer = 1
    reader = 0
20:    proc  1 (user) line  14 "generic.pml" (state 10)
20:    proc  0 (user) line  14 "generic.pml" (state 10)
2 processes created
$```

a verification
(checking a default property: absence of deadlock)

$ spin -a generic.pml
$ gcc -DBFS -o pan pan.c
$ ./pan
pan: invalid end state (at depth 4)
pan: wrote generic.pml.trail
(Spin Version 4.1.0 -- 19 November 2003)
Warning: Search not completed
   + Using Breadth-First Search
   + Partial Order Reduction

Full statespace search for:
   never claim             - (none specified)
   assertion violations    +
   cycle checks            - (disabled by
   -DSAFETY)
   invalid end states      +

State-vector 20 byte, depth reached 4, errors: 1
  44 states, stored
     44 nominal states (stored-atomic)
  16 states, matched
  60 transitions (= stored+matched)
  0 atomic steps
hash conflicts: 0 (resolved)
(max size 2^18 states)

1.253   memory usage (Mbyte)
$
inspection of the error trail

$ spin -t -v generic.pml
  1: proc  1 (user) line   7 "generic.pml" (state 1) [(printer)]
  2: proc  1 (user) line   7 "generic.pml" (state 2) [printer = 0]
  3: proc  0 (user) line  13 "generic.pml" (state 7) [(reader)]
  4: proc  0 (user) line  13 "generic.pml" (state 8) [reader = 0]

spin: trail ends after 4 steps
#processes: 2
    printer = 0
    reader = 0
  4: proc  1 (user) line   8 "generic.pml" (state 3)
  4: proc  0 (user) line  14 "generic.pml" (state 9)

2 processes created
$

process 1

?A

process 2

?B

printer == 0
&&
reader == 0

deadlock
the Spin gui – getting fancy
Spin, Promela, and LTL

• Acronyms:
  – Spin : Simple Promela Interpreter, a nested acronym
  – Promela: Process Meta Language, for behavior specification
  – LTL : Linear Temporal Logic, for property specification

• Spin:
  – model checker generator

• Promela:
  – non-deterministic, guarded command language for specifying the possible system behaviors in a distributed system design
    • systems of interacting, asynchronous threads of execution
  – the purpose is not to prevent the specification of bad or unstructured designs (on the contrary)
    • e.g., gotos are supported
  – the purpose is to allow the specification of designs in such a way that they can be checked with a model checker
context

Promela behavior model

correctness property e.g., in LTL

random and interactive model simulation

guided simulation

SPIN

pan.c model checking code

C compiler

executable model checker

error-trails counter-examples to correctness properties
central concepts

• *finite-state* models only: Promela models are always bounded
  – boundedness in our case guarantees decidability
  – finite state models can still permit infinite executions
• *asynchronous* behavior
  – no hidden global system clock
  – no implied synchronization between processes
• *non-deterministic* control structures
  – to support (inspire?) abstraction from implementation level detail
• *executability* as a core part of the semantics
  – every basic and compound statement is defined by a *precondition* and an *effect*
  – a statement can be executed, producing the *effect*, only when its *precondition* is satisfied; otherwise, the statement is *blocked*
  – *example*: \( q?m \) when channel \( q \) is non-empty, retrieve message \( m \) else block (i.e., wait)
3 types of objects

- processes
- global and local data objects
- message channels
processes

- process behavior is declared in `proctype` declarations
- a `process` is an instantiated `proctype`
- processes can be instantiated in two ways:
  - in the initial system state
    - by adding the prefix `active` to a `proctype` declaration
  - in any other reachable system state
    - with a `run` operator

```plaintext
active [2] proctype eager() {
    run eager();
    run eager()
}
```

2 processes instantiated in initial system state
each process `tries` to instantiate 2 more copies, and then terminates
the proctype eager

```c
active [2] proctype eager()
{
    run eager();
    run eager()
}
```

semi-colons are statement *separators* not statement *terminators*

why is this still a finite model?
run is a Promela *operator*

*run eager( )* is a restricted form of a Promela *expression*
an expression can be used as a *statement* in Promela
run either returns the *pid* of the process it instantiates
or it returns 0 *if no new process can be instantiated*
an expression statement is executable iff it evaluates to non-zero..
the maximum number of active processes is 255 (imposing the bound)
```
proctype irun(byte x)
{
    printf("it is me %d, %d\n", x, _pid)
}

init {
    pid a, b;
    a = run irun(1);
    b = run irun(2);
    printf("I created %d and %d\n", a, b)
}
```

```
$ spin irun.pml
   it is me 1, 1
   I created 1 and 2
      it is me 2, 2
3 processes created
$
```
process interaction and process state

- processes can synchronize their behavior in 2 ways
  - through the use of global (shared) variables
  - via message passing through channels
    - buffered channels or rendezvous channels
  - there is no global ‘clock’ that could be used for synchronization
- each process has its own local state
  - process “program-counter” (i.e., control-flow point)
  - values of all locally declared variables
- the model as a whole has a global state
  - the value of all globally declared variables
  - the contents of all message channels
  - the set of all currently active processes
dynamic process creation

- the *state* of the complete system is maintained in a global state vector
- the *state vector* contains entries for
  - the value of all global variables (including message channels)
  - all active processes
    - each active process containing:
      - the value of all locally declared variables
      - the program counter (the control-flow point)
state vector contains a process stack

- processes are added and deleted in stack (LIFO) order
- a process can start and stop at any time, but it can disappear from the state vector only in LIFO order
- process deletion takes 2 steps: termination and then death
- before a parent can die, all its children must die first…
  - a process pid is only recycled when the process has died
  - an init process always dies last: the first pid can never be recycled
how is finiteness preserved?

• Promela models are necessarily finite-state:
  – there can be only finitely many active processes
  – there can only be finitely many statements in a proctype
  – all data types have a strictly bounded range
    • e.g., the range of a bit or bool is 0..1, the range of a pid or byte is 0..255, the range of a short is $-2^{15}..2^{15}-1$, and the range of an int is $-2^{31}..2^{31}-1$
  – all message channels have a bounded capacity