SPIN: Part 1

15-414/614 Bug Catching: Automated Program Verification

Sagar Chaki
November 12, 2012

What is This All About?

Spin

- On-the-fly verifier developed at Bell-labs by Gerard Holzmann and others
- http://spinroot.com

Promela

- Modeling language for SPIN
- Targeted at asynchronous systems
  - Switching protocols
- http://spinroot.com/spin/Man/Quick.html

History

Work leading to Spin started in 1980

- First bug found on Nov 21, 1980 by Pan
- One-pass verifier for safety properties

Succeeded by

- Pandora (82),
- Trace (83),
- SuperTrace (84),
- SdlValid (88),
- Spin (89)

Spin covered omega-regular properties

Spin Capabilities

Interactive simulation

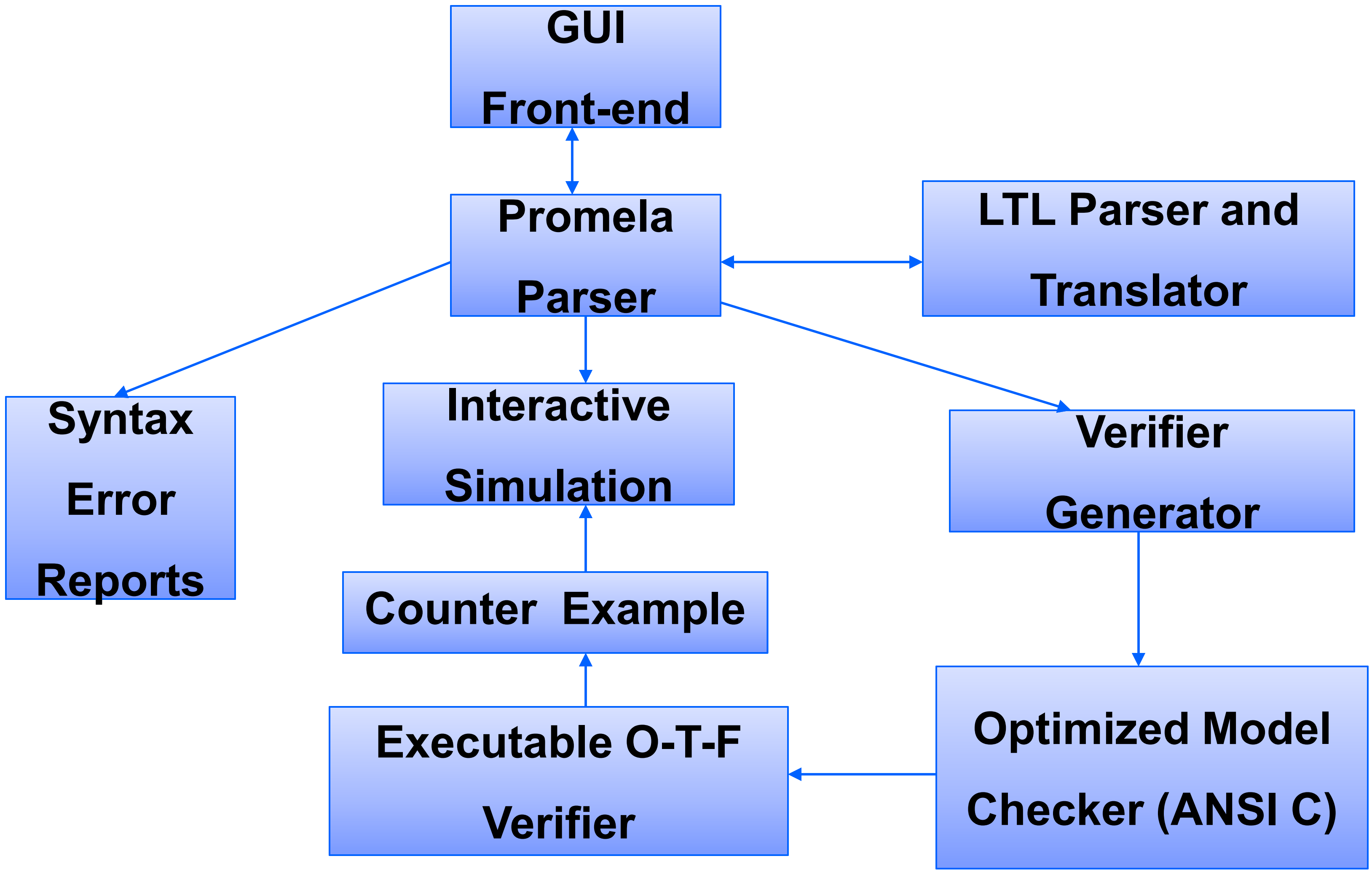
- For a particular path
- For a random path

Exhaustive verification

- Generate C code for verifier
- Compile the verifier and execute
- Returns counter-example

Lots of options for fine-tuning

Spin Overall Structure



Promela

Stands for Process Meta Language

Language for asynchronous programs

- Dynamic process creation
- Processes execute asynchronously
- Communicate via shared variables and message channels
  - Races must be explicitly avoided
  - Channels can be queued or rendezvous
- Very C like

Executability

No difference between conditions and statements

- Execution of every statement is conditional on its executability
- Executability is the basic means of synchronization

Declarations and assignments are always executable

Conditionals are executable when they hold

The following are the same

- while (a != b) skip
- (a == b)

Delimitors

Semi-colon is used as statement separator not a statement terminator

- Last statement does not need semi-colon
- Often replaced by $\rightarrow$ to indicate causality between two successive statements
- (a == b); c = c + 1
- (a == b) $\rightarrow$ c = c + 1

Data Types

Basic : bit/bool, byte, short, int, chan

Arrays: fixed size

- byte state[20];
- state[0] = state[3 * i] + 5 * state[7/j];

Symbolic constants

- Usually used for message types
- mtype = {SEND, RECV};

Process Definition

```
byte state = 2;

proctype A() {
    (state == 1) → state = 3
}

proctype B() {
    state = state – 1
}
```

Process Instantiation

Sample 1

```
byte state = 2;

proctype A() {
    (state == 1) → state = 3
}

proctype B() {
    state = state – 1
}

init { run A(); run B() }
```

run **can be used anywhere**

Sample 2

Process Parameterization

```
byte state = 1

proctype A(byte x; short foo)
{
   (state == 1 && x > 0) → state = foo
}

init { run A(1,3); }
```

Data arrays or processes cannot be passed

Race Condition

Sample 3

```
byte state = 1;

proctype A() {
    byte x = state;
    x = x + 1;
    state = x;
}

proctype B() {
    byte y = state;
    y = y + 2;
    state = y;
}

init { run A(); run B() }
```

Sample 4

Deadlock

```
byte state = 2;

proctype A() {
    (state == 1) → state = state + 1
}

proctype B() {
    (state == 1) → state = state – 1
}

init { run A(); run B() }
```

Sample 5

Atomic sequences

```
byte state = 1;

proctype A() {
   atomic {
      byte x = state;
      x = x + 1;
      state = x;
   }
}
```

```
proctype B() {
   atomic {
      byte y = state;
      y = y + 2;
      state = y;
   }
}

init { run A(); run B() }
```

Message passing

Channel declaration

- chan qname = [16] of {short}
- chan qname = [5] of {byte,int,chan,short}

Sending messages

- qname!expr
- qname!expr1,expr2,expr3

Receiving messages

- qname?var
- qname?var1,var2,var3

Message passing

More parameters sent

- Extra parameters dropped

More parameters received

- Extra parameters undefined

Fewer parameters sent

- Extra parameters undefined

Fewer parameters received

- Extra parameters dropped

Message passing

Sample 6

```
chan x = [1] of {byte,byte};
chan y = [1] of {byte,byte};

proctype A(byte p, byte q)
{
  x!p,q ;
  y?p,q
}
```

```
proctype B() {
  byte p,q;
  x?p,q ; y!q,p
}

init {
  run A(5,7);
  run B()
}
```

Message passing

Convention: first message field often specifies message type (constant)

Alternatively send message type followed by list of message fields in braces

- qname!expr1(expr2,expr3)
- qname?var1(var2,var3)

Executability

Send is executable only when the channel is not full

Receive is executable only when the channel is not empty

Optionally some arguments of receive can be constants

- qname?RECV,var,10
- Value of constant fields must match value of corresponding fields of message at the head of channel queue

len(qname) returns the number of messages currently stored in *qname*

If used as a statement it will be unexecutable if the channel is empty

Composite conditions

Invalid in Promela

- (qname?var == 0)
- (a > b && qname!123)
- Either send/receive or pure expression

Can *evaluate* receives

- qname?[ack,var]

Returns true if the receive would be enabled

Subtle issues

- qname?[msgtype] $\rightarrow$ qname?msgtype
- (len(qname) < MAX) $\rightarrow$ qname!msgtype
- Second statement not necessarily executable after the first
  - Race conditions

Time for example 1

Rendezvous

Channel of size 0 defines a rendezvous port

- Can be used by two processed for a synchronous handshake
- No queueing
- The first process blocks
- Handshake occurs after the second process arrives

Sample 7

Example

```
#define msgtype 33
chan name = [0] of {byte,byte};

proctype A() {
    name!msgtype(99);
    name!msgtype(100)
}

proctype B() {
    byte state;
    name?msgtype(state)
}

init { run A(); run B() }
```

Control flow

We have already seen some

- Concatenation of statements, parallel execution, atomic sequences

There are a few more

- Case selection, repetition, unconditional jumps

Case selection

```
if
:: (a < b) → option1
:: (a > b) → option2
:: else → option3                    /* optional */
fi
```

Cases need not be exhaustive or mutually exclusive

- Non-deterministic selection

Time for example 2

Repetition

```
byte count = 1;
proctype counter() {
        do
        :: count = count + 1
        :: count = count – 1
        :: (count == 0) → break
        od
}
```

Repetition

```
proctype counter()
{
        do
        :: (count != 0) →
                if
                :: count = count + 1
                :: count = count – 1
                fi
        :: (count == 0) → break
        od
}
```

Unconditional jumps

```
proctype Euclid (int x, y)
{
        do
        :: (x > y) → x = x – y
        :: (x < y) → y = y – x
        :: (x == y) → goto done
        od ;
        done:  skip
}
```

Procedures and Recursion

Procedures can be modeled as processes

- Even recursive ones
- Return values can be passed back to the calling process via a global variable or a message

Time for example 3

Timeouts

```
Proctype watchdog() {
    do
    :: timeout → guard!reset
    od
}
```

Get enabled when the entire system is deadlocked

No absolute timing considerations

Assertions

assert(any_boolean_condition)

- pure expression

If condition holds $\Rightarrow$ no effect

If condition does not hold $\Rightarrow$ error report during verification with Spin

Time for example 4

References

http://cm.bell-labs.com/cm/cs/what/spin/

http://cm.bell-labs.com/cm/cs/what/spin/Man/Manual.html

http://cm.bell-labs.com/cm/cs/what/spin/Man/Quick.html

Questions?

Sagar Chaki

Senior Member of Technical Staff

RTSS Program

Telephone: +1 412-268-1436

Email: chaki@sei.cmu.edu

Web

www.sei.cmu.edu/staff/chaki

U.S. Mail

Software Engineering Institute

Customer Relations

4500 Fifth Avenue

Pittsburgh, PA 15213-2612

USA

Customer Relations

Email: info@sei.cmu.edu

Telephone: +1 412-268-5800

SEI Phone: +1 412-268-5800

SEI Fax: +1 412-268-6257