Spin – Model Checker for Promela

Seminar 236800 – Shahar Salzmann
Parallel Algorithms

- Parallel and distributed programming is hard!

- Prone to many errors:
  - Deadlock
  - Livelock, starvation
  - Under specification
  - Over specification
  - ...

• Intro
• Promela
• Spin
• Example
Validating Parallel Algorithms

- **Testing approach**
  - Functionality testing
  - Cover great deal of the code path
  - Cannot cover all code paths
  - Fatal race condition may occur at extremely low probability...

- **Proof of correctness**
  - Not always possible
  - Prone to human error
Validating Parallel Algorithms

Model Checking

“… an automated technique that given a finite state model of a system and a logical property, systematically checks whether this property holds…”

[Clarke & Emerson 1981]

State space

- State machine of interleaved process states

Model checker results

- Property satisfied
- Property unsatisfied + trace to result
Spin & Promela

- **Promela**
  - Concurrent and distributed programming modeling language

- **Spin**
  - General tool for verifying the correctness of distributed software
  - Verification tool for Promela and c

- **Why Promela (and not c)**
  - Modeling allows reducing the program to its most basic states
Today’s Lecture

- **Promela**
  - Understand and get a feel of the modeling language
  - Shared memory example

- **Spin**
  - Overview
  - Verification Example
  - Basic algorithm
  - Optimizations
Promela
Promela

- PROMELA = Process/Protocol Meta Language
- Verification modeling language
- Dynamic creation of concurrent processes
- Communication via message synch/asynch channels
- C-like programming constructs and data types
Promela Example

Definitions

Non Deterministic Choice

Dynamic creation of concurrent processes

Promela Example

(Daniel Kroening, 251-0247-00L Formal Verification lecture slides)
Promela Model Structure

- **Declarations**
  - Type declarations
  - Channel declarations
- **Global variables**
- **Process declarations**
- **Init process**

**Declarations**

```promela
// PROMELA EXAMPLE
// our core street model
//
// global
//
// semaphore
int data;

// process declarations

// Global variables

// process declarations

// Finite State machine
```
Promela Model Structure

- Syntax similar to C and Lotos
  - C like declarations and data-types
  - C like assignments
  - C like structures
  - Lotos like communications
  - Lotos like non-determinism
Declarations

- C-like syntax
- Supports types such as
  - bool
  - int
  - byte
  - short
Data types

- **Enumeration**
  - `mtype = {All possible enumerations};`
  - Can be defined only once

- **User defined types (C structs)**
  - `typedef <name> { types }`
Processes

- Defined just like a C function
  - returns process id
- Actually more like Java Runnable
- Aware of global variables - shared memory

```c
proctype msgChannel(mtype reqType, Id; msg myMsg){
    if
        :: (reqType==SND) -> bus!Id,myMsg;
        :: (reqType==RCV) -> bus?Id,myMsg;
        :: else skip;
    fi;
}
```
Statements

- Make up the body of the process

- C/C++/Java would call them commands
  - But not exactly …

- Statements are either executable/blocked
  - Executable - the statement can be executed immediately
  - Blocked - the statement cannot be executed
Statements

- skip – always executable
- Assert
- C semantics
  - expressions
  - assignments
- Control Flow
  - if/fi
  - do/od
  - break
- Communications
  - send (ch!)
  - receive (ch?)
- Interleaving
  - atomic{…}
  - d_step{…}
- Real-time
  - timeout
Control Flow

- if/fi and do/od structures
  - Guards – condition checking
  - Else – if all fails
  - Skip – do nothing
  - Break – escape from do/od

```c
progtype msgChannel(mtype reqType, id; msg myMsg) {
    if (reqType == SND) Guard
        send(id, myMsg);
    else do
        if (reqType == RCV) if/fi structure boundaries
            bus?id, myMsg;
        else skip;
    fi;
} proctype
```
Control Flow

- Non-deterministic choices
  - First all executable statements are evaluated
  - One is chosen non-deterministically
  - Not really non-deterministic
    - uses random() ...

```plaintext
throw_dice:
  if
    :: skip -> dice=1;
    :: skip -> dice=2;
    :: skip -> dice=3;
    :: skip -> dice=4;
    :: skip -> dice=5;
    :: skip -> dice=6;
  fi
```
Channels

- Message passing between processes
- Synchronous communication
  - [1] of …
  - Handshake protocol
- Asynchronous communication
  - [k] of …
  - K>1
Asynch Channel

Example

Asynch channel

chan bus = [2] of {mtype, msg};

...

proctype msgChannel(mtype reqType, Id; msg myMsg){
    if
        :: (reqType==SND) -> bus!Id,myMsg;
        :: (reqType==RCV) -> bus?Id,myMsg;
        :: else skip;
    fi;
}
Run

- Launch process
- Executable if the process can be launched

Number of processes is limited...

```c
/* first step of computation - compute and send */
state[1] = CORE_PROC1;
state[1] = CORE_SND1;
run msgChannel(SND,ANS1,computation);

/* second step of computation - compute and send */
state[1] = CORE_PROC2;
state[1] = CORE_SND2;
run msgChannel(SND,ANS2,computation);
```
Unless

\{ <statements> \} unless \{ guard ; <statements> \}''
- Execute statements until guard is executable
- Resembles exception handling

```c
proctype MicroProcessor()
{
  {
    ...
  }
  unless { port ? INTERRUPT; ... }
}
```
atomic

- Statements are grouped into an atomic sequence
- No interleaving with other processes
- If a statement is blocked then other processes may perform a step and then statement is rechecked
timeout

- Promela does not have real-time properties!
- Timeout —
  - no process in the model can proceed
  - Global timeout
Assertions

- Defensive programming
- But leaves the code clean ...
  - Consider the following
    ```c
    …
    if (NULL==ptr){
        //clean resources
        …
        exit(1);
    }
    …
    
    - Instead we use
    assert(NULL!=ptr)
    ```
Aiding Readability

- **while**/**for**
  - C/C++/Java programmers can’t do without them...
- **implies** – for logical assertions
- **Use macros/inline:**

```c
#define while(p) {
    do {
        :: (p)->
        #define end_while {
        :: else->break
        od}
```

```c
```
Promela Syntax Summary

- Intuitive to C/C++/Java programmers
- Easily model large systems
  - No need to think about implementation details
- inline / macro tricks aid readability
Interleaving Semantics

- Each process is modeled into states
- Processes are interleaved to generate the statespace
- Statespace grows very quickly
  - numStatesA X numStatesB X numStatesC X ...

• Intro
• Promela
• Spin
• Example
Interleaving Semantics

Process A

1
2
3

Process B

1
2
3

Interleaved Statespace

(1,1) → (2,1) → (3,1) → (3,2) → (2,2) → (1,2)

(1,1) → (2,2) → (3,2) → (3,3) → (2,3) → (1,3)
d_step

- Same semantics as atomic
- No intermediate states are constructed
- If statement is blocked a runtime error is generated
- May be used to reduce the size of the statespace
**d_step**

Interleaved Statespace

Process A | Process B
---|---
1 | 1
2 | 2
3 | 3

(1,1) -> (3,1) -> (3,3) -> (3,1) -> (1,3) -> (3,3) -> (3,1)
Spin
Spin

- **SPIN = Simple Promela Interpreter**
- Model checker for Promela
- Open source
- One of the most powerful model checkers (Holzman 1991)
- Verifies
  - Safety
  - Liveness
  - LTL properties
Spin

- Lots of references
  - Books
  - Online help
  - Workshops

check out:
http://spinroot.com/spin/whatispin.html#X
Applications of Spin

- Flood barrier Rotterdam
- Call processing logic – Lucent (formerly AT&T)
- Mission critical SW of space missions
  - Cassini - handoff algorithms for dual CPU control
  - Mars exploration rover – motor resource manager
XSpin

- Graphical interface for Spin
- Generates spin commands based on menu selections
- Simulator for Promela
- Spin runs in the background
XSpin as Model Simulator

- Intro
- Promela
- Spin
- Example

Simulator output

Global variable values

Sequence Chart
XSpin as Model Verifier

LTL Formulae Verification

Verification Results

Error example simulation

Suggested Action
Optionally, repeat the run with a different search depth to find a shorter path to the error.
Or, perform a GUIDED simulation to retrace the error found in this run, and skip the first series of steps if the error was found at a depth greater than about 100 steps.

Setup Guided Simulation.. Run Guided Simulation.. Close

Intro
Promela
Spin
Example
XSpin Verification

Intro

Promela

Spin

Example

Safety
Spin Verification

- **Assert Progress** using special labels
  - **end** — valid end states (otherwise may be recognized as deadlock)
  - **progress** — mark states that have to be visited in every execution cycle
  - **accept** — states that cannot be part of an infinite execution cycle
Spin Output

Intro
Promela
Spin
Example

Verification Results

Search Parameters

Memory Usage

Static Analysis

Warnings: for p.o. reduction to be valid the never claim must be stutter-invariant

(never claims generated from LTL formulae are stutter-invariant)

depth 0: Claim reached state 5 (line 101)
depth 12: Claim reached state 9 (line 106)
depth 50: Claim reached state 9 (line 106)

(Spin Version 5.1.6 -- 9 May 2006)
+ Partial Order Reduction

Full statespace search for:
never claim +
assertion violations + (if within scope of claim)
acceptance cycles + (fairness disabled)
invalid end states - (disabled by never claim)

State-vector 88 byte, depth reached 55, errors: 0
228 states, stored (335 visited)
301 states, matched
633 transitions = visited + matched
0 atomic steps
hash conflicts: 0 (resolved)

Stats on memory usage (in Megabytes):
0.022 equivalent memory usage for states (stored)=state-vector + overhead)
0.269 actual memory usage for states (unsuccessful compression: 1281.10%)
state-vector as stored = 1327 byte + 16 byte overhead
2.000 memory used for hash table (~w19)
0.305 memory used for DFS stack (~m 10000)
2.501 total actual memory usage

unreached in proctype msgChannel
line 21, "pan ___", state 4, "bus?ld,myMsg data"
(1 of 9 states)
unreached in proctype driver
(0 of 9 states)
unreached in proctype core
line 78, "pan __", state 11, "end"
(1 of 11 states)
unreached in proctype unit:
(0 of 4 states)
Behind The Curtains

- SPIN does not actually do any work!
  - Generates c sources for a problem-specific model checker
- Saves memory
- Improves performance
- Allows direct insertion of c into the model
Behind The Curtains

- **Step 1 — create model**
  - Translate model into a state machine

- **Step 2 — create checker**
  - Generate never claim from checking parameters
  - Translate never claim into a state machine

- **Step 3 — create model checker**
  - Translate state machines to c (pan.c)
  - Generate c for DFS searches of illegal states

*** Now we have a problem specific model checker ***

- **Step 4 — verify model**
  - Build pan.c to pan.exe and run
Building the model

- Each process is translated into a state machine
- All of the states of the different processes are interleaved
- End product – a state machine representing the valid model transitions.
The model states

- Each state represents a unique model snapshot

- Contains
  - Global values
  - Channel contents
  - For each process
    - Locals
    - PC
Verification Algorithm

- No tricks here!
  - Simple concept
  - Clever optimizations

- DFS through the model searching for illegal states
- Verification is performed on-the-fly

DFS(state s)
assert(legal(s))
for each successor t of s which we haven’t visited
  DFS(t)
Statespace explosion...

- **DFS requires state matching**

- **Each visited state must be saved**

- **Spin provides several algorithms for statespace compression**
Memory Management

- State compression challenge
  - Good compression vs. Quick state comparison
- Quick state comparison requires that all states be compressed in the same way
  - Dynamic Huffman encoding - X
  - Lempel Ziv - X
  - Run length coding - √
- The greater the compression the greater the penalty
State Compression

- As of 1995 release a new compression method was released
- Knowledge of what we are compressing can help
- Every process and every channel have a relatively small number of states
- Large number of states is usually a result of process interaction

**Solution:**
- Store every process state
- Store every channel state
- Global model state uses pointers to process/channel states
State Matching

- Naïve – O(n)

- Solution - Hash states
  - Entire state space may be huge
  - Only small percentage is actually reachable
  - Good hash function will prevent conflicts

- Spin uses multi bit-state hashing
Bit State Hashing

- Allocate huge array
- Each bit represents a valid state
- Set bit if state is used
- Use hash function
  - Map small sub-range of huge range
  - Quick access
- Problem: no collision detection
- Solution (partial) — use 2 bits and 2 hash functions
Other Optimizations

- Partial order reduction
  - ??

- Minimized automaton
  - Encode States

- Dataflow analysis
  - Dead variables/code

- Slicing algorithm
  - Throw away unneeded branches
C verification with spin

- **C commands within Promela model**
  - `c_decl`, `c_state`, `c_code`
  - Converter from c to Promela

- **I/O commands don’t really make sense…**

- **No dynamic memory**

- **Process interleaving – not the real level of atomicity**

- **State space explosion even a bigger problem**
Example
Core-Driver

- Systems which are inherently concurrent

- Used in real time systems
  - Embedded systems
  - Multiple core systems
  - Communications

- Pure SW models
  - DB access
  - RMI
Core-Driver

- **Core**
  - Separate resource
  - responsible for fast optimized execution
  - Close to the HW
  - Requires complex interface

- **Driver**
  - Interface with the core
  - Used by high level components to utilize the core
Our Core-Driver Protocol

- Communication via bus
- **Driver** sends request
- Work split between 2 cores
- **Driver2Core** communication on separate line
- **Core2Driver** communication on bus
Our Core

- Idle State: Wait for requests
- Computation #1: Compute some of the required request
- Receive Request
- Finish C2/Send(C2)
Our Driver

Start:
Driver is activated

Wait #1:
Wait for first computation result

Wait #2:
Wait for the rest of the result

Finish:

Send(Request,Driver2CoreA),
Send(Request,Driver2CoreB)

Result #1 received

Result #2 received
Summary

- **Promela**
  - Syntax – all a c/c++/java programmer needs is a cheat sheet
  - Interleaving semantics – attempt any possible interleaving

- **Spin**
  - Simulator/Model checker
  - Safety/Liveness/Weak Fairness/LTL
  - Thorough checking at cost of memory

- **Model Checking**
  - Space Efficiency vs. Runtime Performance
Class Experience - Mutual Exclusion

- Load a model of a simple process

- Initial verification using LTL formula
  - Find mutual exclusion violation

- Fix process definition
  - Verify fix
References

- http://lwn.net/Articles/243851/
- http://www.cs.technion.ac.il/~ssdl/veritech/tools/core_2_spin/about.htm
- http://www.labri.fr/perso/fleury/courses/FDS05/courses.html
Questions?