FORMAL METHODS IN NETWORKING
COMPUTER SCIENCE 598D, SPRING 2010
PRINCETON UNIVERSITY

LIGHTWEIGHT MODELING
IN PROMELA/SPIN AND ALLOY

Pamela Zave

AT&T Laboratories—Research
Florham Park, New Jersey, USA
LIGHTWEIGHT MODELING

DEFINITION

- constructing a very abstract model of the core concepts of a system
- using a "push-button" analysis tool to explore its properties

"analysis" is more general than "verification"

WHY IS IT "LIGHTWEIGHT"?

- because the model is very abstract in comparison to a real implementation, and focuses only on core concepts, it is small and can be constructed quickly
- because the analysis tool is "push-button", it yields results with little effort

WHAT IS ITS VALUE?

- it is a design tool that reveals conceptual errors early
- it is a documentation tool that provides complete, consistent, and unambiguous information to implementors and users
- it is easy (at least to get started) and fun!

"If you like surprises, you will love lightweight modeling."
—Pamela Zave

in contrast, theorem proving is not "push-button"

Read introduction to Software Abstractions for Daniel Jackson's view.
WHY IS LIGHTWEIGHT MODELING EASY, SURPRISING?

EASY + SURPRISING = FUN

PROGRAMMING:
1 write a program
2 think of a test case
3 run the program on the test that you thought of

LIGHTWEIGHT MODELING
1 write a model (no bigger than a small program)
2 push the "analyze" button
3 get results from all possible executions in a particular category, including "tests" you would never have thought of!

HOW MODEL CHECKERS DO IT

all data structures have fixed size, so state space is bounded (includes implicit structures such as call stack)

initial state

compute all possible state transitions

continue expanding graph until a fixed point is reached

the result is an explicit, finite reachability graph representing all possible states, state transitions, and executions (finite or infinite paths through the graph)
WHAT IS THE HIDDEN CHALLENGE?

It is so easy to write a model, ask the analyzer a question, get an answer . . .

. . . but not so easy to know what any of these means in the real world.

STATEMENTS IN MODEL

- domain knowledge: description of the environment in which the system will operate (fact or assumption)

- specification: an implementable description of how the hardware/software system should behave

- requirement: a description of how the environment should behave when the system is implemented and deployed

- sanity check: intended to be redundant

NONDETERMINISM IN MODEL

- environment choice
- implementation freedom
- system failure
- concurrency

ANALYSIS QUESTIONS

- Is the model consistent (can be realized) ?
- Does the model mean what I think it means ("validation") ?
- Is the model correct ("verification") ?

sanity checks help

Read "Deriving specifications from requirements: An example" for an example with all the parts.
SPIN AND PROMELA

SPIN IS A MODEL CHECKER

- originated in the 1980's at Bell Labs
- freely available and actively maintained
- well-engineered and mature
- large user base, in both academia and industry
- used in mission-critical and safety-critical software development
- Spin user workshops have been held annually since 1995

PROMELA IS ITS MODELING LANGUAGE

- unlike most mature model checkers, Spin is intended for software verification, not hardware verification
- "Promela" derived from "protocol modeling language"
- Promela resembles a primitive programming language
- it has built-in message queues for inter-process communication

Spin and other model checkers can also be used for verification of implementations, although that is not the focus here

Read CalTech lecture for Holzmann's introduction to model checking.
mtype = \{ invite, accept, reject \}
chan left = [3] of \{mtype\};
chan right = [3] of \{mtype\};
proctype caller (chan in, out) {
    out!invite;
    inviting: do
        :: in?accept; goto confirmed
        :: in?reject; goto end
    od;
    confirmed: do
        :: in?invite; out!accept
        :: out!invite; in?accept
    od;
    end: skip
}
proctype callee (chan in, out) {
    in?invite;
    invited: do
        :: out!accept; goto confirmed
        :: out!reject; goto end
    od;
    confirmed: do
        :: in?invite; out!accept
        :: out!invite; in?accept
    od;
    end: skip
}
init { atomic { run caller(left,right);
    run callee(right,left)
    } }
mtype = \{ invite, accept, reject \}
chan left = [3] of \{mtype\};
chan right = [3] of \{mtype\};
proctype caller (chan in, out) {
    out!invite;
    inviting: do
        :: in?accept; goto confirmed
        :: in?reject; goto end
    od;
    confirmed: do
        :: in?invite; out!accept
        :: out!invite; in?accept
    od;
    end: skip
}
proctype callee (chan in, out) {
    in?invite;
    invited: do
        :: out!accept; goto confirmed
        :: out!reject; goto end
    od;
    confirmed: do
        :: in?invite; out!accept
        :: out!invite; in?accept
    od;
    end: skip
}
init { atomic { run caller(left,right);
    run callee(right,left)
    } }
SIP VERSION 2

FIXES DEADLOCK DISCOVERED IN VERSION 1

mtype = { invite, accept, reject, race }

proctype caller (chan in, out) {
    out!invite;
    inviting: do
        :: in?accept; goto confirmed
        :: in?reject; goto end
    od;
    confirmed: do
        :: in?invite; out!accept
        :: out!invite; goto reInviting
    od;
    reInviting: do
        :: in?accept; goto confirmed
        :: in?race; goto confirmed
        :: in?invite; out!race
    od;
    end: skip
}

proctype callee (chan in, out) {
    in?invite;
    invited: do
        :: out!accept; goto confirmed
        :: out!reject; goto end
    od;
    confirmed: do
        :: in?invite; out!accept
        :: out!invite; goto reInviting
    od;
    reInviting: do
        :: in?accept; goto confirmed
        :: in?race; goto confirmed
        :: in?invite; out!race
    od;
    end: skip
}

until further notice, we are using only default analysis in Spin

neither process terminates, but analysis reports no errors because it is only looking for invalid end states
SIP VERSION 3

ADDS BYE AND ITS ACK TO END DIALOG

mtype = { invite, accept, reject, race, bye, byeAck }

proctype caller (chan in, out) {
    out!invite;
    inviting:   do
        :: in?accept; goto confirmed
        :: in?reject; goto end
        od;
    confirmed:  do
        :: in?invite; out!accept
        :: in?bye; out!byeAck; goto end
        :: out!invite; goto reInviting
        :: out!bye; goto end
        od;
    reInviting: do
        :: in?accept; goto confirmed
        :: in?race; goto confirmed
        :: in?invite; out!race
        od;
    end:         skip
}

proctype callee (chan in, out) {
    in?invite;
    invited:    do
        :: out!accept; goto confirmed
        :: out!reject; goto end
        od;
    confirmed:  do
        :: in?invite; out!accept
        :: in?bye; out!byeAck; goto end
        :: out!invite; goto reInviting
        :: out!bye; goto end
        od;
    reInviting: do
        :: in?accept; goto confirmed
        :: in?race; goto confirmed
        :: in?invite; out!race
        od;
    end:         skip
}
if one of the processes is relInviting, and the first message in its input queue is bye, it will be blocked forever
proctype caller (chan in, out) {
    out!invite;
    inviting:   do
                 :: in?accept; goto confirmed
                 :: in?reject; goto end
                 od;
    confirmed:  do
                 :: in?invite; out!accept
                 :: in?bye; out!byeAck;
                  goto end
                 :: out!invite; goto relInviting
                 :: out!bye; goto end
                 od;
    relInviting: do
                 :: in?invite; out!race
                 :: in?accept; goto confirmed
                 :: in?race; goto confirmed
                 :: in?bye; out!byeAck;
                      goto end
                 od;
    end:        skip
}
```plaintext
SIP VERSION 4

proctype caller (chan in, out) {
    out!invite;
    inviting: do
        :: in?accept; goto confirmed
        :: in?reject; goto end
    od;
    confirmed: do
        :: in?invite; out!accept
        :: in?bye; out!byeAck;
        goto end
        :: out!invite; goto reInviting
        :: out!bye; goto end
    od;
    reInviting: do
        :: in?invite; out!race
        :: in?accept; goto confirmed
        :: in?race; goto confirmed
        :: in?bye; out!byeAck;
        goto end
        od;
    end: skip
}

proctype callee (chan in, out) {
    in?invite;
    invited: do
        :: out!accept; goto confirmed
        :: out!reject; goto end
    od;
    confirmed: do
        :: in?invite; out!accept
        :: in?bye; out!byeAck;
        goto end
        :: out!invite; goto reInviting
        :: out!bye; goto end
    od;
    reInviting: do
        :: in?invite; out!race
        :: in?accept; goto confirmed
        :: in?race; goto confirmed
        :: in?bye; out!byeAck;
        goto end
        od;
    end: skip
}

if a process sends a bye and ends, it may leave messages unread and unprocessed

"-q" runtime option makes an end state invalid if it has nonempty queues
```
SIP VERSION 5

GUARANTEES THAT BOTH PROCESSES ARE INPUT-ENABLED

proctype caller (chan in, out) {
  out!invite;
  inviting: do
    :: in?invite; assert(false)
    :: in?accept; goto confirmed
    :: in?reject; goto end
    :: in?race; assert(false)
    :: in?bye; assert(false)
    :: in?byeAck; assert(false)
  od;
  confirmed: do
    :: in?invite; out!accept
    :: in?accept; assert(false)
    :: in?reject; assert(false)
    :: in?race; assert(false)
    :: in?bye; out!byeAck;
    goto end
    :: in?byeAck; assert(false)
  od
  reInviting: do
    :: in?invite; out!race
    :: in?accept; goto confirmed
    :: in?reject; assert(false)
    :: in?race; goto confirmed
    :: in?bye; out!byeAck;
    goto end
    :: in?byeAck; assert(false)
  od;
  Byeing: do
    :: in?invite
    :: in?accept; assert(false)
    :: in?reject; assert(false)
    :: in?race; assert(false)
    :: in?bye; out!byeAck
    :: in?byeAck; goto end
    :: in?byeAck; goto confirmed
    :: in?byeAck; assert(false)
    :: in?byeAck; goto end
  od;
  end: skip
}
proctype caller (chan in, out) {
    out!invite;
    inviting: do
        :: in?accept; goto confirmed
        :: in?reject; goto end
    od;
    confirmed: do
        :: in?invite; out!accept
        :: in?bye; out!byeAck;
            goto end
        :: out!invite; goto relInviting
        :: out!bye; goto Byeing
    od;
    relInviting: do
        :: in?invite; out!race
        :: in?accept; goto confirmed
        :: in?race; goto confirmed
        :: in?bye; out!byeAck;
            goto end
    od;
    Byeing: do
        :: in?invite
        :: in?bye; out!byeAck
        :: in?byeAck; goto end
    od;
    end: skip
}

proctype callee (chan in, out) {
    in?invite;
    invited: do
        :: out!accept; goto confirmed
        :: out!reject; goto end
    od;
    confirmed: do
        :: in?invite; out!accept
        :: in?bye; out!byeAck;
            goto end
        :: out!invite; goto relInviting
        :: out!bye; goto Byeing
    od;
    relInviting: do
        :: in?invite; out!race
        :: in?accept; goto confirmed
        :: in?race; goto confirmed
        :: in?bye; out!byeAck;
            goto end
    od;
    Byeing: do
        :: in?invite
        :: in?bye; out!byeAck
        :: in?byeAck; goto end
    od;
    end: skip
}