CSCI 234

Design of Internet Protocols: PROMELA and SPIN

George Blankenship

Outline

- Verification and Validation
- History and motivation
- Spin
- Promela language
- Promela model

Verification vs. Validation

- Software verification is often confused with software validation
- Software verification is a verification of conformance to the specification
- Software validation is a validation of the compliance with the requirements

Common Design Flaws

- Deadlock
- Livelock
- Underspecification
- Overspecification
- Violations of constraints
- Assumptions about speed

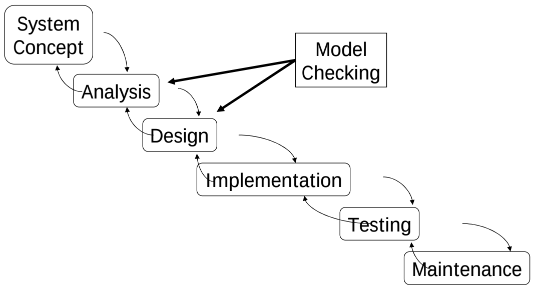
Diagnosing Design Flaws

- Complexity makes design flaws difficult to uncover.
- Engineers often use simplified models (prototypes) for design verification.
- Abstract models can also be used to verify concurrent systems.

What is Model Checking?

- Use a simplified model of our system.
- Verify the system exhaustively.
- Automatically check that given properties hold in all possible states.

System Development



PROMELA and SPIN George Blankenship 7

System State Based Analysis

- Complete state space must be represented
- State space defined by significant variables
- Each integer variable has 2^{32} distinct possibilities. Two such variables have 2^{64} possibilities.
- In concurrent protocols, the number of states usually grows exponentially with the number of processes.

PROMELA and SPIN George Blankenship 8

State Space Capture

- System is the asynchronous composition of processes
- For each state the successor states are enumerated using the transition relation of each process

PROMELA and SPIN George Blankenship 9

Reducing Complexity

- Problem: state space explosion!
- Automatic state space compression and reduction by SPIN.
- Manual reduction techniques by the designer.
  - We need to find the smallest sufficient model of our system.
  - Biggest challenge!

If it is so constrained, is it of any use?

- Many protocols are finite state.
- Many programs or procedure are finite state in nature. Can use abstraction techniques.
- Sometimes possible to decompose a program, and prove part of it by model checking and part by theorem proving.
- Many techniques to reduce the state space explosion (Partial Order Reduction).

Alternating Bit Protocol

```
mtype = {MSG, ACK};
chan toS = [2] of {mtype, bit};
chan toR = [2] of {mtype, bit};
proctype sender(chan in, chan out)
{
   bit sendbit, recvbit;
   do
   :: out!MSG, sendbit ->
        in?ACK, recvbit
        if
        :: recvbit == sendbit ->
           sendbit = 1-sendbit
        :: else
        fi
   od
}
```

```
proctype receiver(chan in, chan out)
{
   bit recvbit;
   do
   :: in?MSG, recvbit ->
        out!ACK, recvbit
   od
}

init
{
   run sender(toS, toR);
   run receiver(toR, toS);
}
```

What is this all about?

- SPIN
  - On-the-fly verifier developed at Bell-labs by Gerard Holzmann and others
- Promela
  - Modeling language for SPIN
  - Targeted at asynchronous systems

PROMELA and SPIN George Blankenship 13

"First Computer Bug"

PROMELA and SPIN George Blankenship 14

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)

PROMELA and SPIN George Blankenship 15

SPIN

- SPIN (Simple PROMELA Interpreter)
  - Tool for analyzing the logical consistency of concurrent systems.
  - Takes a PROMELA model as input.
- Model-checker.
- Based on automata theory.
- Allows LTL or automata specification
- Efficient (on-the-fly model checking, partial order reduction).
- Developed in Bell Laboratories.

PROMELA and SPIN George Blankenship 16

SPIN Features

- "press on the button" verification (model checker)
- efficient implementation
- graphical user interface (Xspin)
- used for research and industry
- contains more than two decades research on advanced computer aided verification (many optimization algorithms)

PROMELA and SPIN George Blankenship 17

The language of SPIN

- Called Promela
- The expressions are from C.
- The communication is from CSP.
- The constructs are from Guarded Command.

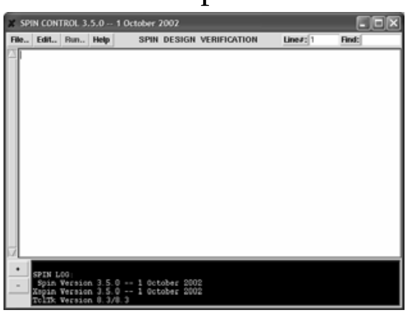
PROMELA and SPIN George Blankenship 18

Command Line Tools

- Spin
  - Generates the Promela code for the LTL formula
  - Generates the C source code
- Pan (Process Analyzer)
  - Performs the verification
    - Has many compile time options to enable different features
    - Optimized for performance

PROMELA and SPIN George Blankenship 19

Xspin



PROMELA and SPIN George Blankenship 20

Simulator

- Spin can also be used as a simulator
  - Simulated the Promela program
- It is used as a simulator when a counterexample is generated
  - Steps through the trace
  - The trace itself is not “readable”
- Can be used for random and manually guided simulation as well

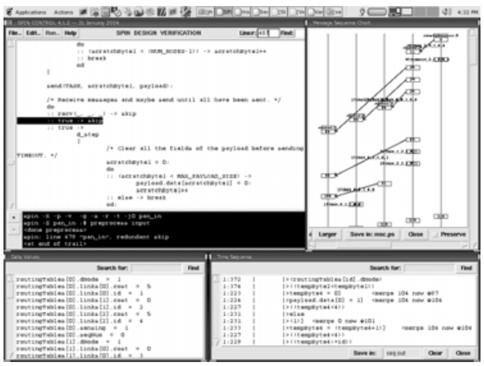
PROMELA and SPIN George Blankenship 21

XSpin Features

- Graphical front-end to the SPIN model checker.
- Features:
  - Editor
  - Syntax checking
  - Simulation
  - Verification
  - Requirements specification

PROMELA and SPIN George Blankenship 22

XSpin Screenshot



23

Types of Properties

- Invalid end-states (deadlock)
- Assertion violations
- Unreachable code
- Liveness properties
  - Non-progress cycles
  - Acceptance cycles
- Linear Temporal Logic (LTL) formulae

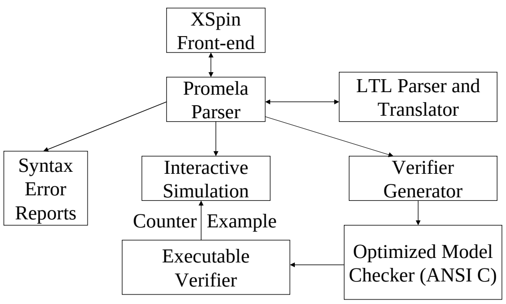
PROMELA and SPIN George Blankenship 24

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

PROMELA and SPIN George Blankenship 25

Spin overall structure



PROMELA and SPIN George Blankenship 26

PROMELA

- PROMELA (Process/Protocol Meta Language)
  - Specification language to model finite-state systems.
  - Dynamic creation of concurrent processes.
  - Communication via synchronous or asynchronous message channels.
  - Non-deterministic (you'll see!).
- 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

PROMELA and SPIN George Blankenship 27

Finite Systems Only!

- No unbounded data.
- No unbounded message channels.
- No unbounded processes.
- No unbounded process creation.

Variables and Types

- Five different (integer) basic types.
- Arrays
- Records (structs)
- Type conflicts are detected at runtime
- Default initial value of basic variables (local and global) is 0.

Variables

- Variables should be declared
- Variables can be given a value by:
  - assignment
  - argument passing
  - message passing
- Variables can be used in expressions
- Most arithmetic, relational, and logical operators of C/Java are supported

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};

Basic Types (integer)

Declarations	Value range
bit turn=1;	[0..1]
bool flag;	[0..1]
byte counter;	[0..255]
short s1, s2;	[-2^{16}-1.. 2^{16}–1]
int msg;	[-2^{32}-1.. 2^{32}–1]

Arrays

- <type> <array name>[<array size];
- byte a[27]; // array a can hold 27 bytes
- bit flags[4]; // array flags can hold 4 bits
- Same as C/C++
- Array index starts at 0
- Array index ends at size-1

Expressions

- Arithmetic: +, -, *, /, %
- Comparison: >, >=, <, <=, ==, !=
- Boolean: &&, ||, !
- Assignment: =
- Increment/decrement: ++, --

Records

- Type definition defines records (structure)

```
typedef MyRecord {
short f1; byte  f2;
}
```

- Variable declaration defines variable

```
MyRecord rr;
```

- Values are reference field by field

```
rr.f1 = …
rr.f2 = …
byte a = rr.f2;
```


Message types and channels

- Message type is enumeration declaration <type name>={<value list>}
- mtype = {OK, READY, ACK, ERROR}
  - #define OK = 1;
  - #define READY=2;
  - #define ACK=3
  - #define ERROR=4
- mtype Mvar = ACK

Channels

- Channel defines queue that is used to pass messages between processes
  chan <name>=[<size>] of {<message record>}
- chan Ng=[2] of {mtype, byte, byte},
  Next=[0] of {byte}
- Enqueue a message – blocks if channel has no space
  <name>!<record fields>
- Dequeue a message – blocks if channel is empty
  <name>?<variables to receive fields>
- Ng!OK(5,4); /* send message type OK with data 5 and 4
  Ng!OK,5,4; /* same as OK(5,4)
- Ng?OK(byte0,byte1); /* store data if type is OK

PROMELA and SPIN George Blankenship 37

Delimiters

- Semi-colon is used a statement separator not a statement terminator
- Last statement does not need semi-colon
- Often replaced by **->** to indicate causality between two successive statements

(a == b); c = c + 1
(a == b) -> c = c + 1

PROMELA and SPIN George Blankenship 38

Statements

- The body of a process consists of a sequence of statements
- A statement is either
  - executable: the statement can be executed immediately
  - blocked: the statement cannot be executed
- Executable/blocked depend on the global state of the system.

PROMELA and SPIN George Blankenship 39

Executable Statements

- An assignment is always executable
- An expression is also a statement; it is executable if it evaluates to non-zero
  - 2 < 3 always executable
  - x < 27 only executable if value of x is smaller 27
  - 3 + x executable if x is not equal to –3

PROMELA and SPIN George Blankenship 40

Executability

- The body of a process consists of a series of statements.
- Statements are either executable or blocked.
- 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 always executable
- Conditionals are executable when they hold
- The following are the same

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

```
(x < y)

i = 3

ch!MSG

ch?msg
```

PROMELA and SPIN George Blankenship 41

Statements

- The skip statement is always executable
- A run statement is only executable if a new process can be created (the number of processes is bounded)
- A printf statement is always executable
- Statements in a sequence are separated by a semi-colon: “;”
- A given statement in a sequence isn’t executable until previous statement executed

PROMELA and SPIN George Blankenship 42

Sample

```
int x;
proctype A( ) {
  int y=1;
  skip;
  run N( );
  x=2;
  x>2 && y==0;
  skip;
}
```

Executable if N can be created... (points to: run N( );)

Can only become executable if some other process makes x greater than 2 (points to: x>2 && y==0;)

PROMELA and SPIN George Blankenship 43

assert(<expression>);

- The assert statement is always executable
- If <expr> evaluates to zero, SPIN will exit with an error, as the <expr>“has been violated”
- The assert statement is often used to check whether certain properties are valid in a state
- proctype monitor() { assert(n <= 3); }
- proctype receiver() { ...
  toReceiver ? msg;
  assert(msg !=ERROR);
  ... }

PROMELA and SPIN George Blankenship 44

if-statement

if :: choice$_1$ -> stat$_{1.1}$; stat$_{1.2}$; stat$_{1.3}$; ...
:: choice$_2$ -> stat$_{2.1}$; stat$_{2.2}$; stat$_{2.3}$; ...
:: ...
:: choice$_n$ -> stat$_{n.1}$; stat$_{n.2}$; stat$_{n.3}$; ...
fi;

PROMELA and SPIN George Blankenship 45

if-statement

- If there is at least one $choice_i$ (guard) executable, the if statement is executable and SPIN non-deterministically chooses
- If no $choice_i$ is executable, the if-statement is blocked
- The operator "->"is equivalent to ";"
- The else guard is always executable
- Guard need not be exhaustive or mutually exclusive

PROMELA and SPIN George Blankenship 46

do-loops

do
:: $choice_1$ -> $stat_{1.1}$; $stat_{1.2}$; $stat_{1.3}$; ...
:: $choice_2$ -> $stat_{2.1}$; $stat_{2.2}$; $stat_{2.3}$; ...
:: ...
:: $choice_n$ -> $stat_{n.1}$; $stat_{n.2}$; $stat_{n.3}$; ...
od;

PROMELA and SPIN George Blankenship 47

do-loops

- With respect to the choices, a do statement behaves in the same way as an if statement
- However, instead of ending the statement at the end of the chosen list of statements, a do-statement repeats the choice selection
- The (always executable) break statement exits a do-loop statement and transfers control to the end of the loop

PROMELA and SPIN George Blankenship 48

goto-statement

- Transfer control to a non-sequential statement
  goto <label >;
- Transfers execution to label
- Each Promela statement might be labeled
- Quite useful in modeling communication protocols

PROMELA and SPIN George Blankenship 49

Interleaving Semantics

- Promela processes execute concurrently
- Non-deterministic scheduling of the processes
- Processes are interleaved
- All statements are atomic; each statement is executed without interleaving with other processes
- Each process may have several different possible actions enabled at each point of execution

PROMELA and SPIN George Blankenship 50

atomic-statement

- Groups statements into an atomic sequence
  atomic{ st_1; st_2; ... st_n }
- all statements are executed in a single step (no interleaving with statements of other processes)
- is executable if st_1 is executable
- if a st_i is blocked, the “atomicity token” is (temporarily) lost and other processes may do a step

PROMELA and SPIN George Blankenship 51

PROMELA Model Basics

- Promela model consists of:
  - type declarations
  - channel declarations
  - variable declarations
  - process declarations

```
mtype = {MSG, ACK};
chan toS;
chan toR;
bool flag;

proctype sender()
{
  …
}
proctype receiver()
{
  …
}
```

PROMELA and SPIN George Blankenship 52

Promela Code

```
mtype = {MSG, ACK};
chan toS = ...
chan toR = ...
bool flag;
proctype Sender() {
 ...
}
proctype Receiver() {
...
}
init {
 ...
 }
```

process body

creates processes

PROMELA and SPIN George Blankenship 53

Promela Model Syntax

- Type declarations
  - mtype, typedefs, constants
- Channel declarations
  - chan ch= [dim] of {type, ...}
  - asynchronous: dim> 0
  - rendez-vous: dim== 0
- Global variable declarations
  - can be accessed by all processes
- Process declarations
  - behaviourof the processes
  - local variables + statements
- [initprocess]
  - initializes variables and starts processes

PROMELA and SPIN George Blankenship 54

Processes

- A process is defined by a proctype definition
- A process executes concurrently with all other processes, independent of speed of behavior
- A process communicates with other processes
  - using global (shared) variables
  - using channels
- There may be several processes of the same type
- Each process has its own local state

PROMELA and SPIN George Blankenship 55

Process

name

```
proctype Sender(chan in; chan out) {
    bit sndB, rcvB;
    do
    :: out ! MSG, sndB ->
            in ? ACK, rcvB;
            if
            :: sndB == rcvB -> sndB = 1-sndB
            :: else -> skip
            fi
    od
}
```

body

formal parameters

local variables

The body consist of a sequence of statements

PROMELA and SPIN George Blankenship 56

Process Example

byte state = 2;

proctype A() { (state == 1) -> state = 3 }

proctype B() { state = state – 1 }

PROMELA and SPIN George Blankenship 57

Process Instantiation

byte state = 2;
proctype A() { (state == 1) -> state = 3 }
proctype B() { state = state – 1 }
init { *run* A(); *run* B() }

- *run* can be used anywhere

Parameter passing

proctype A(byte x; short foo) {
(state == 1) -> state = foo
}
init { run A(1,3); }

- Data arrays or processes cannot be passed

Variable scoping

- Global scope variables are known throughout the model
- Process local scope variables are only known within the process
- Parameters are only known within the process and initialized to passed value

- byte foo, bar, baz;
  proctype A(byte foo) {
  byte bar;
  baz = foo + bar;
}

Races and deadlock

```
byte state = 1;
proctype A() {
   (state == 1) -> state = state + 1
}
proctype B() {
   (state == 1) -> state = state – 1
}
init { run A(); run B() }
```


Atomic Sequence

```
byte state = 1;
proctype A() { atomic {
   (state == 1) -> state = state + 1
}  }
proctype B() { atomic {
   (state == 1) -> state = state – 1
}  }
init() { run A(); 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)
- 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 Mismatch

- 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 Example

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

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

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

init { run A(1,2); run B(3,4) }
```


Executability

- Send is executable only when the channel is not full
- Receive is executable only when the channel is not empty
- A channel size reflects the ability of a channel to “store” a message for a future consumer
- 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

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

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

Timeouts

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

- timeout is a predefined global boolean that is set true when the entire system is deadlocked
- No absolute timing considerations

Processes

- Process are created using the run statement (which returns the process id)
- Processes can be created at any point in the execution (within any process)
- Processes start executing after the run statement.
- Processes can also be created by adding active in front of the proctype declaration
  - Parameters will be initialized to 0

PROMELA and SPIN George Blankenship 70

Processes

- There may be several processes of the same proctype
- Each process has its own local state:
  - process counter(location within the proctype) – contents of the local variables

PROMELA and SPIN George Blankenship 71

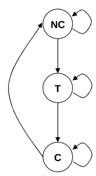
Hello World!

```
active proctype Hello( ) {
   printf("Hello process, my pid is: %d\n",_pid);
}
init{
   int lastpid;
   printf("init process, my pidis: %d\n", _pid);
   lastpid= run Hello();
   printf("last pid was: %d\n",lastpid);
 }
```

PROMELA and SPIN George Blankenship 72

Concurrent Processes

```
mtype = { NONCRITICAL, TRYING, CRITICAL};
show mtype state[2];
proctype process(int id) {
   beginning:
   noncritical:
        state[id] = NONCRITICAL;
        if
                :: goto noncritical;
                :: true;
        fi;
   trying:
        state[id] = TRYING;
        if
                :: goto trying;
                :: true;
        fi;
   critical:
        state[id] = CRITICAL;
        if
                :: goto critical;
                :: true;
        fi;
        goto beginning;}
init { run process(0); run process(1); }
```



PROMELA and SPIN George Blankenship 73

Message Type Definition

```
mtype = { NONCRITICAL, TRYING, CRITICAL};
show mtype state[2];
proctype process(int id) {
   beginning:
   noncritical:
        state[id] = NONCRITICAL;
        if
                :: goto noncritical;
                :: true;
        fi;
   trying:
        state[id] = TRYING;
        if
                :: goto trying;
                :: true;
        fi;
   critical:
        state[id] = CRITICAL;
        if
                :: goto critical;
                :: true;
        fi;
        goto beginning;}
init { run process(0); run process(1); }
```

PROMELA and SPIN George Blankenship 74

XSpin Directive (show)

```
mtype = { NONCRITICAL, TRYING, CRITICAL};
show mtype state[2];
proctype process(int id) {
   beginning:
   noncritical:
        state[id] = NONCRITICAL;
        if
                :: goto noncritical;
                :: true;
        fi;
   trying:
        state[id] = TRYING;
        if
                :: goto trying;
                :: true;
        fi;
   critical:
        state[id] = CRITICAL;
        if
                :: goto critical;
                :: true;
        fi;
        goto beginning;}
init { run process(0); run process(1); }
```

PROMELA and SPIN George Blankenship 75

Process Definition

```
mtype = { NONCRITICAL, TRYING, CRITICAL};
show mtype state[2];
proctype process(int id) {
   beginning:
   noncritical:
        state[id] = NONCRITICAL;
        if
                :: goto noncritical;
                :: true;
        fi;
   trying:
        state[id] = TRYING;
        if
                :: goto trying;
                :: true;
        fi;
   critical:
        state[id] = CRITICAL;
        if
                :: goto critical;
                :: true;
        fi;
        goto beginning;}
init { run process(0); run process(1); }
```

PROMELA and SPIN George Blankenship 76

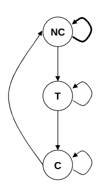
Start Execution

```
mtype = { NONCRITICAL, TRYING, CRITICAL};
show mtype state[2];
proctype process(int id) {
   beginning:
   noncritical:
        state[id] = NONCRITICAL;
        if
                :: goto noncritical;
                :: true;
        fi;
   trying:
        state[id] = TRYING;
        if
                :: goto trying;
                :: true;
        fi;
   critical:
        state[id] = CRITICAL;
        if
                :: goto critical;
                :: true;
        fi;
        goto beginning;}
init { run process(0); run process(1); }
```

PROMELA and SPIN George Blankenship 77

Execution

```
mtype = { NONCRITICAL, TRYING, CRITICAL};
show mtype state[2];
proctype process(int id) {
   beginning:
   noncritical:
        state[id] = NONCRITICAL;
        if
                :: goto noncritical;
                :: true;
        fi;
   trying:
        state[id] = TRYING;
        if
                :: goto trying;
                :: true;
        fi;
   critical:
        state[id] = CRITICAL;
        if
                :: goto critical;
                :: true;
        fi;
        goto beginning;}
init { run process(0); run process(1); }
```



PROMELA and SPIN George Blankenship 78

Enabled Statements

- A statement needs to be enabled for the process to be scheduled.

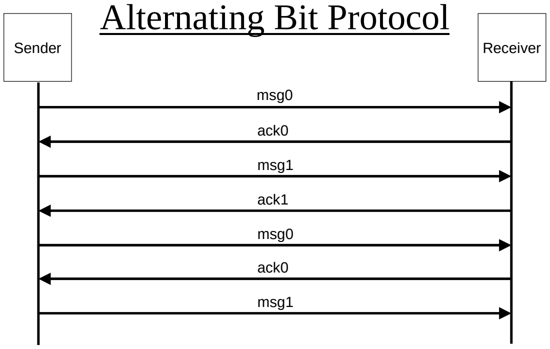
```
bool a, b;
proctype p1(){
  a = true;
  a & b;
  a = false;
}
proctype p2(){
  b = false;
  a & b;
  b = true;
}
init { a = false; b = false; run p1(); run p2(); }
```

These statements are enabled only if both ***a*** and ***b*** are true.

This statements can never be enabled since *b* is always false.

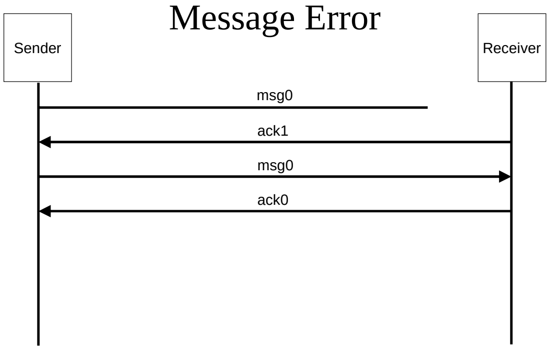
PROMELA and SPIN George Blankenship 79

Alternating Bit Protocol



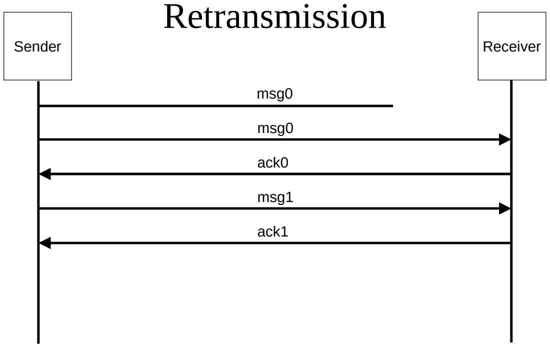
PROMELA and SPIN George Blankenship 80

Message Error



PROMELA and SPIN George Blankenship 81

Retransmission



PROMELA and SPIN George Blankenship 82

Message and Channel Definitions

```
mtype = { msg0, msg1, ack0, ack1 }
chan sender = [1] of { mtype };
chan receiver = [1] of { mtype };
```

PROMELA and SPIN George Blankenship 83

Sender Process

```
active proctype Sender(){
  do
  ::
    if
    :: receiver?msg0;
    :: skip
    fi;
    do
    :: sender?ack0 -> break
    :: sender?ack1
    :: timeout ->
        if
        :: receiver!msg0;
        :: skip
        fi;
    od;
```

```
  ::
    if
    :: receiver?msg1;
    :: skip
    fi;
    do
    :: sender?ack1 -> break
    :: sender?ack0
    :: timeout ->
        if
        :: receiver!msg1;
        :: skip
        fi;
    od;
  od;
}
```

PROMELA and SPIN George Blankenship 84

Receiver Process

```
active proctype Receiver(){
  do
  ::
    do
    :: receiver?msg0 -> sender!ack0; break;
    :: receiver?msg1 -> sender!ack1
    od

    do
    :: receiver?msg1 -> sender!ack1; break;
    :: receiver?msg0 -> sender!ack0
    od
  od
}
```

PROMELA and SPIN George Blankenship 85

Lynch's Protocol

- … a reasonable looking but inadequate scheme …
- Full duplex operation on two channels
- If previous reception was error-free, the next message on reverse channel contains ACK, otherwise NAK
- If previous reception carried NAK or was in error, retransmit, other wise send new message

PROMELA and SPIN George Blankenship 86

Lynch's Protocol Problems

- Cannot send ACK/NAK without data – need to send fill
- Startup not defined – send error to start process
- Receiver cannot tell whether transmission is retransmission of properly received data or new data

PROMELA and SPIN George Blankenship 87