

TUTORIAL

DESIGN AND VALIDATION OF PROTOCOLS

ICNP93, 19 OCTOBER 1993

Gerard J. Holzmann
Computing Science Research Center

AT&T Bell Laboratories
600 Mountain Ave 2C-521
Murray Hill, NJ 07974
USA

TUTORIAL

DESIGN AND VALIDATION OF PROTOCOLS

INTRODUCTION I.

BASIC ELEMENTS OF A PROTOCOL II.

BUILDING VALIDATION MODELS III.

EXPRESSING CORRECTNESS REQUIREMENTS IV.

AUTOMATED VALIDATION V.

FTP ACCESS TO SPIN SOURCE

```
ftp research.att.com
login: anonymous
Password: yourname@yourmachine
-> cd netlib/spin
-> binary
-> mget *
...
```

EMAIL

Internet USA:	netlib@research.att.com
Internet Oslo:	netlib@nac.no
EARN/BITNET:	netlib%nac.no@norunix.bitnet
X.400:	s=netlib; o=nac; c=no;
EUNET/uucp:	nac!netlib
Australia:	netlib@draci.cs.uow.edu.au

EXAMPLE

```
mail netlib@research.att.com
send index
.
```

DOCUMENTATION

``Design and Validation of Computer Protocols''
Gerard J. Holzmann
Prentice Hall, Englewood Cliffs, NJ, USA, ©1991

HELP

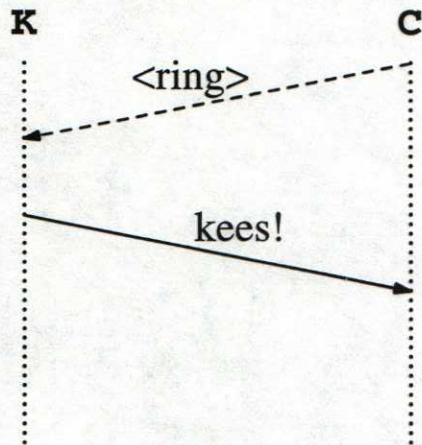
```
mail gerard@research.att.com
```

BASIC ELEMENTS OF A PROTOCOL II.

- INFORMAL EXAMPLE 1.
- PURPOSE OF PROTOCOLS 2.
- FIVE BASIC ELEMENTS 3.
- THE DESIGN PROBLEM 4.

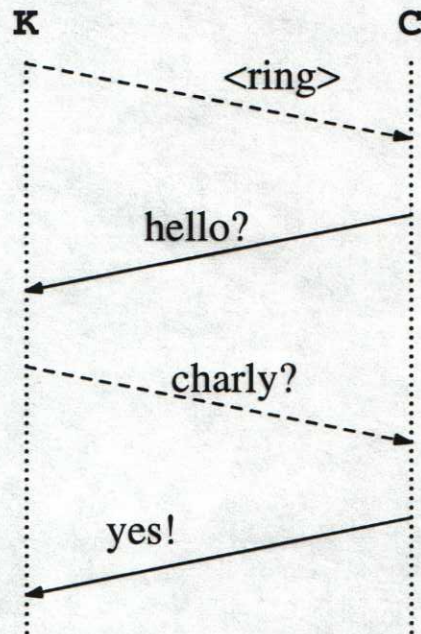
PHONE PROTOCOLS

(USA->Netherlands)

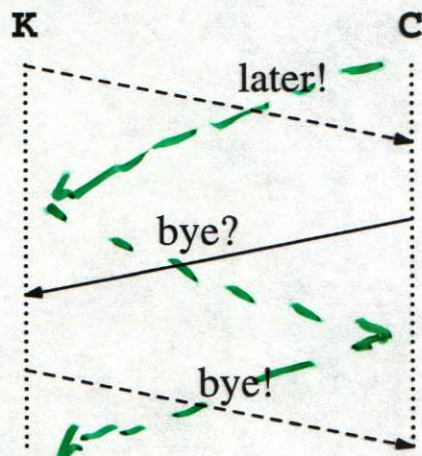


Initialization

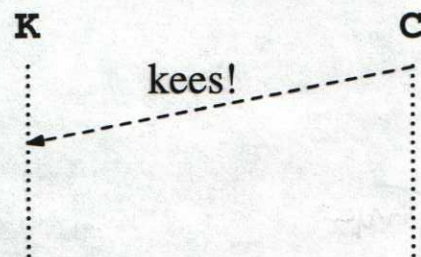
(Netherlands->USA)



Initialization



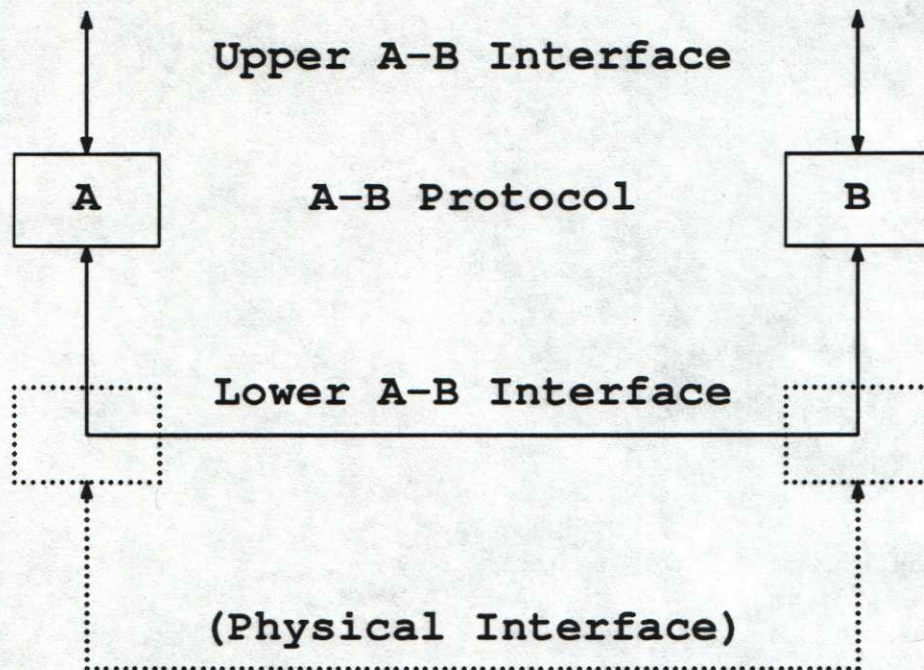
Termination



Termination

Sender

Receiver

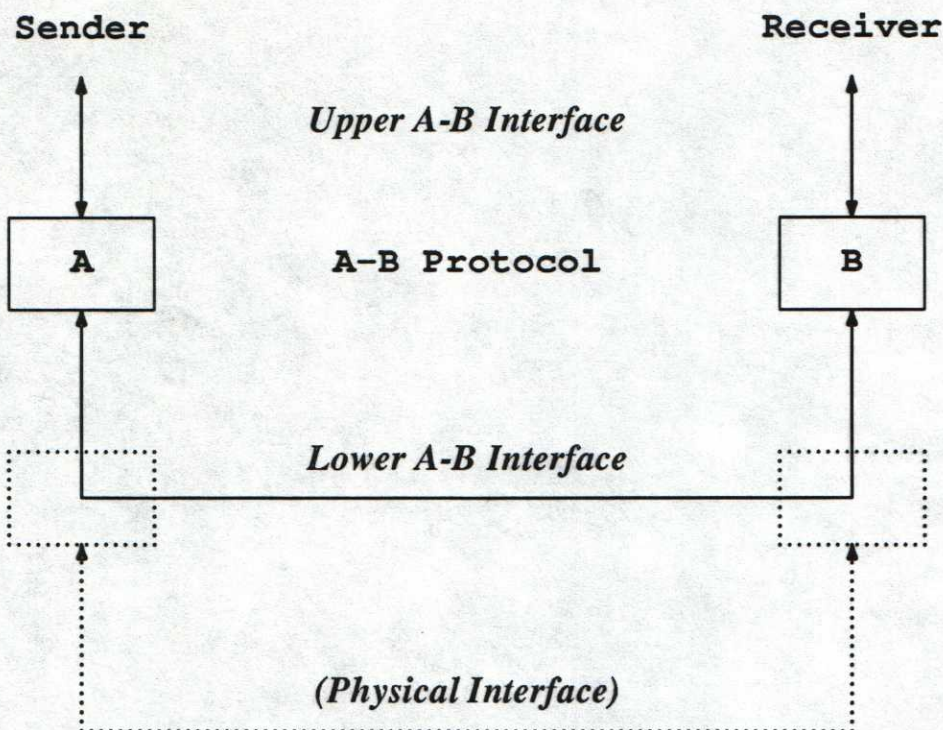


Levels of Abstraction

THE FIVE BASIC ELEMENTS OF A PROTOCOL

1. The *service* to be provided by the protocol (UI)
2. The *constraints* of the environment (LI)
3. The *vocabulary* of messages used
4. The *encoding* of messages (syntax/format)
5. The *procedure rules* guarding message exchange (grammar/behavior)

back to layer structure
vii.2d



Signals
bits
bytes
fields
messages
dialogues

Levels of Abstraction

- Given are the characteristics of a specific *Lower Interface* and the definition of an abstract *Upper Interface*
- *LI* = e.g., an optical fiber, a packet layer protocol.
UI = e.g., error-free file transfer, data-base service
- Design problem: transform *LI* step by step into *UI*
- *LI* together with the specification of the process that transforms it into *UI*, has the properties of a language
 - vocabulary
 - syntax
 - grammar
- To define a protocol is to define that language
Omitting any language element leads to an incomplete or ambiguous specification

EXAMPLE

1 The protocol defines a simplex data-transfer channel, i.e., it
2 will pass data in only one direction, from sender to receiver.
3 Control information, however, flows in both directions. It is
4 assumed that the system has perfect error detection.

5 To each message sent from A to B we attach an extra bit called
6 the alternation bit. After B receives the message it decides if
7 the message is error-free. It then sends back to A a
8 verification message, consisting of a single verify bit,
9 indicating whether or not the immediately preceding A to B
10 message was error-free. After A receives this verification, one
11 of three possibilities hold:

12 1. The A to B message was good

13 2. The A to B message was bad

14 3. A cannot tell if the A to B message was good or bad
15 because the verification message (sent from B to A) was in error
16 In cases 2 and 3 A resends the same A to B message as before. In
17 case 1 A fetches the next message to be sent, and sends it,
18 inverting the setting of the alternation bit with respect to the
19 previous A to B message.

20 Whenever B receives a message that is not in error it compares
21 the alternation bit of this new message to the alternation bit of
22 the most recent error-free reception. If the alternation bits
23 are equal the new message is not accepted. The new message is
24 accepted only if the two alternation bits differ. The
25 verification messages from B to A indicate error-free reception
26 independently of the acceptance of the messages.

27 Initialization of this scheme depends upon A and B agreeing on an
28 initial setting of the alternation bit. This is accomplished by
29 an A to B message whose error-free reception (but not necessarily
30 acceptance) forces B's setting of the alternation bit. Multiple
31 receptions of such a message cannot do harm.

32 This protocol has the property that every message fetched by A is
33 received error-free at least once and accepted at most once by B.

THE PROTOCOL DESIGN PROBLEM

The heart of the protocol design problem is the design of a consistent set of procedure rules (a protocol grammar)

- We must be able to express design decisions and formal requirements in a notation that is
 - Unambiguous
 - Implementation Independent
- We need reliable tools for checking the consistency of the design decisions against formal requirements
- Decisions on, for instance, message format definitions, the layout of bits and fields in messages, should be *deferred* until we have found a set of procedure rules that can guarantee that the correctness requirements are met
- With such a method, we could formalize both:
 - Lynch's example specification
 - Its correctness requirements (lines 32 and 33)

and prove that the correctness claims are justified

THE PROTOCOL DESIGN PROBLEM

A mature Engineering Discipline:

- Discriminates between Requirements and Implementations
- Uses Engineering Models (Prototypes) to verify design decisions
- Can predict Essential Characteristics of products *before* they are Implemented

How can we give protocol engineering these characteristics?

~~How can we give protocol engineering these characteristics?~~

line 1 - 4	UI and LI	Conventions
5 - 10	Format (+ procedure)	Ext. Syntax
10 - 31	Procedure Rules	Grammar
32 - 33	Requirements / service ?	Service.

PROTOCOL VALIDATION MODELS III.

- THE LANGUAGE PROMELA 1.
- MODEL OF AN IDEAL CHANNEL 2.
- MODEL OF A NON-IDEAL CHANNEL 3.
- MODEL OF SENDER AND RECEIVER 4.
- THE INITIAL PROMELA PROCESS 5.
- EXAMPLES 6.

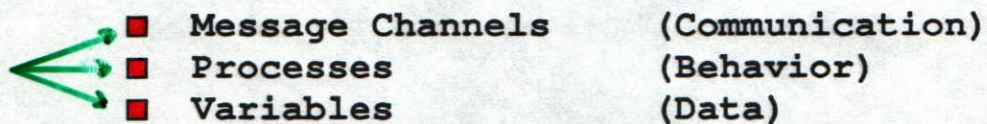
PROTOCOL PROTOTYPES: VALIDATION MODELS

- Models are *abstractions*
They are representations of yet unbuilt things.
- Models contain *less detail* than the object that is modeled.
- Models should contain only the details needed to verify design requirements
 - The detail depends on the requirements expressed
- Models should be effectively verifiable

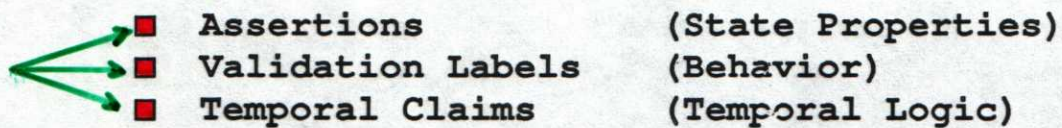
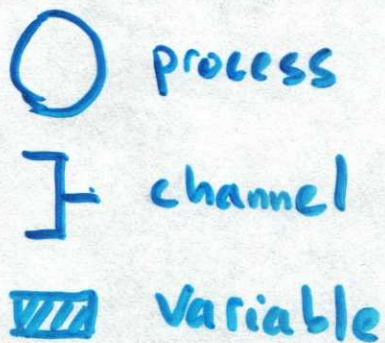
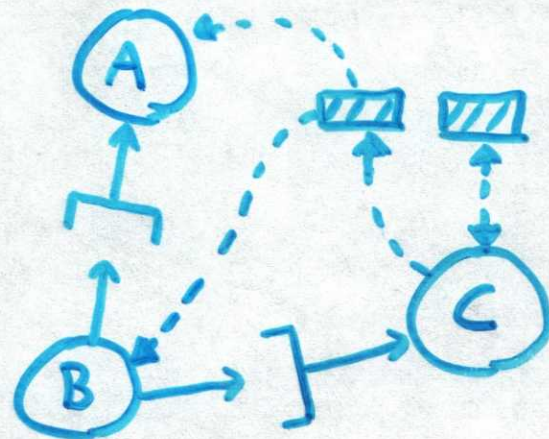
PROMELA - PROTOCOL META LANGUAGE

- A proto-typing language for describing validation models and the corresponding correctness requirements.

- There are three types of abstract objects:



- There are three types of correctness requirements.



PROMELA OBJECTS

- Channels -- (A) Synchronous Communication

```
chan fromA;                                /* Declare */  
fromA = [N] of { byte, byte }; /* Create */
```

- Proctypes -- Asynchronous Behavior

```
proctype lower(...) { ... }               /* Declare */  
run lower(...);                           /* Create */
```

- Variables -- Data

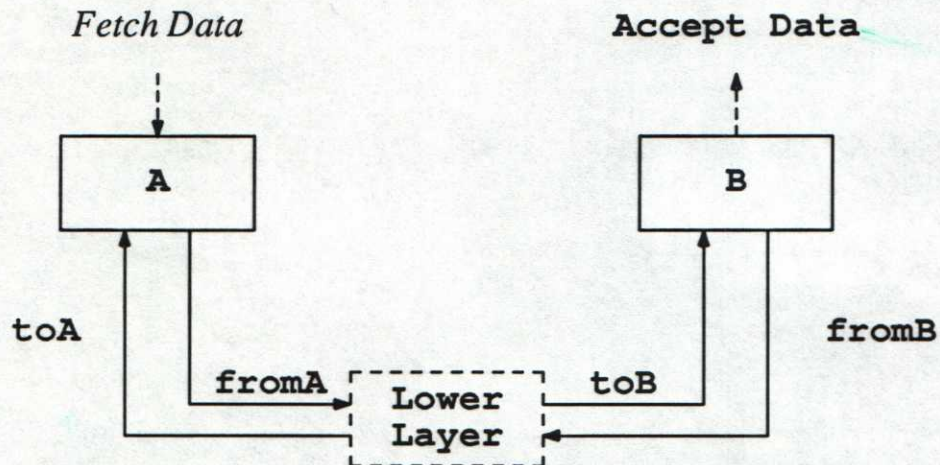
```
int a=4; byte d; bit b=0;                 /* Both */
```

Table A.2 — Basic Data Types

Name	Size (bits)	Usage
bit	1	unsigned
bool	1	unsigned
byte	8	unsigned
short	16	signed
int	32	signed

EXAMPLE: AN IDEAL TRANSMISSION CHANNEL

(No Message Loss, Distortion, Insertion, Reordering)



A-B protocol

(Boxes are Processes – Arrows are Channels)

```
1  proctype lower(chan fromA, toA, fromB, toB)
2  {    byte d; bit b;
3
4      do
5      :: fromA?data(d,b) -> toB!data(d,b)
6      :: fromB?control(b) -> toA!control(b)
7      od
8  }
```

SEND AND RECEIVE

Sample Declarations

```
mtype { data, control }; /* enum values 1 .. 2 */
chan fromA = [2] of { byte, byte, bit }; /* data, udata, seqno */
chan toB = [2] of { byte, byte, bit }; /* data, udata, seqno */
byte d; bit b;
```

Send

Expr!Expr[,Expr]* where Expr = any expression

Example

toB!data,d,b == toB!data(d,b)

toB -- channel id
data -- a constant
d,b -- variable names

Semantics

Parameter *expressions* evaluated upon sending

Receive

Expr?Name[,Name]* where Name = variable or constant

Example

fromA?data,d,b == fromA?data(d,b)

fromA -- channel id
data -- a constant
d,b -- variable names

Semantics

Constants in a receive parameter list are *matched*
Variables are *set* after a match on the remainder

RULES FOR EXECUTABILITY

Synopsis

- Every statement is either Executable or Non-Executable (i.e., blocking)
- A *Send* is Executable iff the destination channel is Non-Full
- A *Receive* is Executable iff the channel is Non-Empty
- An *Assignment* is always Executable
- A *Condition* is Executable iff it holds (evaluates to True)
- The Evaluation of a Condition may not have Side-Effects
- A *Selection* is Executable iff at least one guard is Executable
- If more than one guard is Executable one is selected at random

Example

```
if
  :: fromA?data,b,d
  :: toB!data(b,d)
  :: (a == 0) -> ...
  :: (a > 0) -> ...
  :: (a > N) -> break
fi
```

EXAMPLE: NON-IDEAL TRANSMISSION CHANNEL

(Distortion and Loss, No Insertion or Reordering)

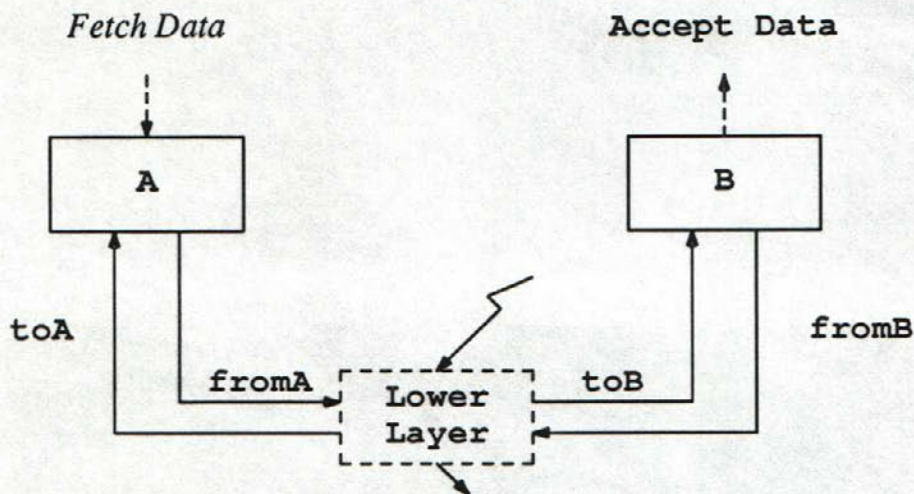


Figure 2 - A-B protocol

(Boxes are Processes - Arrows are Channels)

```
0  mtype = { data, control, error }
1  proctype lower(chan fromA, toA, fromB, toB)
2  {   byte d; bit b;
3
4      do
5          :: fromA?data(d,b) ->
6              if
7                  :: toB!data(d,b)           /* correct */
8                  :: toB!error(d,b)         /* distorted */
9                  :: skip                    /* lost mesg */
10             fi
11         :: fromB?control(b) ->
12             if
13                 :: toA!control(b)
14                 :: toA!error(b)
15                 :: skip
16             fi
17         od
18 }
```

VALIDATION MODEL OF SENDER

```
1  proctype A(chan in, out)
2  {   byte mt;           /* message data          */
3     bit  at;           /* alternation bit      */
4     bit  vr;           /* verify bit           */
5
6     FETCH;             /* get a new mesg       */
7     out!data(mt,at);    /* send it              */
8     do
9     :: in?control(vr) -> /* line 11, await response */
10        if
11        :: (vr == 1) -> /* line 12, correct send */
12            FETCH;      /* line 17, new message */
13            at = 1-at    /* line 18, toggle bit */
14        :: (vr == 0) -> /* line 13, send error */
15            skip         /* line 16, don't fetch */
16        fi;
17        out!data(mt,at) /* line 16              */
18    :: in?error(vr) -> /* line 14-15, recv error */
19        out!data(mt,at) /* line 16              */
20    od
21 }
```

To send a sequence of integers in the data field:

```
#define FETCH      mt = (mt+1)%MAX
```

Notes

- Semicolons are statement separators
They are not statement terminators
- A 'skip' statement is always executable
It has no effect when executed
- A process of type 'A' does not terminate

VALIDATION MODEL OF RECEIVER

```
1  proctype B(chan in, out)
2  {   byte mr;           /* message data          */
3     byte last_mr;       /* mr of last error-free msg */
4     bit  ar;            /* alternation bit           */
5     bit  lar;           /* ar of last error-free msg */
6
7     do
8       :: in?error(mr,ar) ->           /* lines 7-10    */
9         out!control(0)                /* lines 8,25,26 */
10      :: in?data(mr,ar) ->             /* lines 7-10    */
11        out!control(1);               /* lines 8,25,26 */
12        if                            /* line 20        */
13          :: (ar == lar) ->            /* line 21        */
14            skip                       /* line 23        */
15          :: (ar != lar) ->           /* line 24        */
16            ACCEPT;                   /* line 24        */
17            lar = ar;                 /* line 22        */
18            last_mr = mr
19        fi
20    od
21 }
```

``The same message cannot be accepted twice in a row''
(line 33 from the Lynch's specification):

```
#define ACCEPT  assert(mr == (last_mr+1)%MAX)
```

Notes

- An 'assert' statement is executable only if the assertion holds. It is an error (correctness violation) if the assertion fails.
- A process of 'proctype B' does not terminate.

THE INITIAL PROCESS (1)

- Every Validation Model must contain an 'init' section.

- The smallest Validation Model is

```
init { skip }
```

- The smallest model that does something useful (?)

```
init { printf("hello world\n") }
```

- A Complete 'init' for Sender, Receiver, and Channel.

```
1  #define N    2
2  #define MAX  8
3  #define FETCH      mt = (mt+1)%MAX
4  #define ACCEPT     assert(mr == (last_mr+1)%MAX)
5
6  mtype = { data, control, error };
7
8  #include "lynch0.A"
9  #include "lynch0.B"
10 #include "lynch0.C"
11
12 init {
13     chan fromA = [N] of { byte, byte, bit };
14     chan toB   = [N] of { byte, byte, bit };
15     chan fromB = [N] of { byte, bit };
16     chan toA   = [N] of { byte, bit };
17
18     atomic {
19         run A(toA, fromA);
20         run B(toB, fromB);
21         run lower(fromA, toA, fromB, toB)
22     }
23 }
```

THE INITIAL PROCESS (2)

Notes

- Run statements return immediately after instantiating a copy of a previously declared 'proctype'
run A(toA, fromA);
- Run statements are executable only if a process of the right type can indeed be instantiated.
- All statements in an 'atomic' sequence are executed in one indivisible step.
- An atomic statement is executable only if its first statement is executable
- It is an error if an atomic statement blocks
- A process can terminate only when all its children have terminated first
- The 'init' process blocks in its end-state until all processes have terminated

EXAMPLE - MUTUAL EXCLUSION

```
/* Peterson's solution - 1981 */

#define true    1
#define false   0

bool flag[2];
bool turn;

proctype user(bool i)
{
    flag[i] = true;
    turn = i;
    (flag[1-i] == false || turn == 1-i);
crit:  skip; /* critical section */
    flag[i] = false
}

proctype mutex()
{
    assert(!user[1]:crit || !user[2]:crit)
}

init
{
    atomic {
        run user(0);
        run user(1);
        run mutex()
    }
}
```

EXAMPLE - RECURSION

```
/** Print an integer number n in any base b */

#define NUMBER 17779 /* any integer */
#define BASE 2 /* any base >= 2 */
#define ZERO 48 /* ASCII value of character 0 */

proctype printn(int n; byte b; chan a)
{
    int m;

    m = n/b;
    if
    :: m > 0 ->
        chan done = [0] of { byte };
        run printn(m, b, done);
        done?0 /* wait until done */
    :: m == 0 ->
        skip
    fi;
    printf("%c", ZERO + n-m*b);
    a!0
}

init {
    chan done = [0] of { byte };
    run printn(NUMBER, BASE, done);
    done?0; /* wait until done */
    printf("\n")
}

/* Output:
*
* $ spin recursion
* 100010101110011
* 16 processes created
* $
*/
```

AN EFFECTIVE VALIDATION METHOD

- SINGLE LANGUAGE APPROACH
 - Prove Equivalence of Behaviors
- DUAL LANGUAGE APPROACH
 - Prove Consistency of Behavior vs Requirements
- BOTH ARE COMBINATORIAL PROBLEMS
 - Fundamental P-SPACE Complexity

AN UNAMBIGUOUS NOTATION

TWO SCHOOLS

- THE SINGLE LANGUAGE APPROACH
 - CCS, LOTOS, ... Z, VDM,
 - Specify Multiple Versions of Behavior
- THE DUAL LANGUAGE APPROACH
 - PROMELA, S/R, (SDL+, ...)
 - Specify Behavior + Requirements

CORRECTNESS REQUIREMENTS IV.

- GENERAL 1.
 - Formalizing Behavior
- ASSERTIONS 2.
 - Process Invariants
 - System Invariants
- VALIDATION LABELS 3.
 - End-State
 - Progress-State
 - Acceptance-State
- TEMPORAL CLAIMS 4.
 - Never Primitive
 - Matching Behavior
 - Linear Time Temporal Logic

CORRECTNESS REQUIREMENTS – GENERAL

- There are no *correct* protocols in any absolute sense.
- Protocols can only be proven correct with respect to specific correctness requirements.
- Some requirements are protocol independent, e.g. absence of
 - *Deadlock* (i.e., improper termination)
 - *Unspecified Reception* (i.e., incompleteness)
 - *Dead Code* (i.e., overspecification)
 - *Non-Progress Cycles*
 - *Buffer-Overrun*

 - Self-Stabilization
 - ...
- Some requirements are protocol dependent:
 - Absence of *Assertion Violations*
 - Performance of *Service*
- A validation is meaningless if the wrong requirements are checked. (i.e., proving absence of deadlock does *not prove* the performance of a service, e.g. effective data transfer).

EXPRESSING CORRECTNESS REQUIREMENTS

1. Semantics of Model

State Space = Complete Set of Potential Behaviors

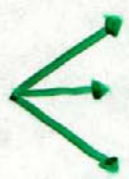
2. Types of Requirements

A. Statements about (Subsets) of State Space

B. Statements about Paths through State Space

- Terminating Paths
- Cyclic Paths

3. Mechanism

- 
- A. Assertions and End-State Labels
 - B. Non-Progress and Acceptance Labels
 - C. Temporal Claims

CORRECTNESS REQUIREMENTS – ASSERTIONS

- The statement **assert(condition)** is always executable and can be placed anywhere.
- The condition is an arbitrary Boolean expression.
- If the condition is true, the statement has no effect.
- It is an error if there is at least one execution sequence in which the condition can be false when the assertion is reached.

EXAMPLE 1. PROCESS INVARIANTS

```
byte state = 1;

proctype A()
{
    (state == 1) -> state = state + 1;
    assert(state == 2)
}

proctype B()
{
    (state == 1) -> state = state - 1;
    assert(state == 0)
}

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

EXAMPLE 2. SYSTEM INVARIANTS

```
proctype monitor()
{
    assert(state >= 0 && state <= 2)
}
...
init { run monitor(); run A(); run B() }
```

CORRECTNESS REQUIREMENTS – VALIDATION LABELS (1)

— END STATE LABELS —

- By default, the final state in a terminating execution sequence must satisfy two criteria to be considered valid:
 - All message channels are empty
 - Every process that was instantiated has reached either the end of its code, or a state marked as an **endstate**
- An end-state label is any label-name with prefix **end**

EXAMPLE BINARY SEMAPHORE

```
mtype = { p, v, ... }  
  
chan sema = [0] of { byte };  
  
proctype dijkstra()  
{  
end:    do  
        :: sema!p -> sema?v  
        od  
}  
  
proctype user()  
{  
end:    do  
        ::      sema?p;  
           /* critical section */  
           sema!v;  
           /* non-critical section */  
        od  
}
```

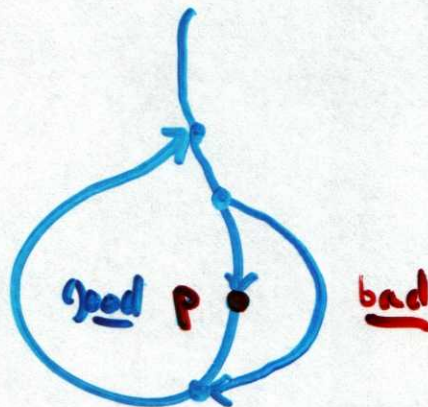
CORRECTNESS REQUIREMENTS – VALIDATION LABELS (2)

— PROGRESS STATE LABELS —

- A *Non-Progress Cycle* is a finite sequence of statements that can be repeated infinitely often without achieving **progress**
- The user can define what constitutes **progress**
For instance
 - Incrementing a Sequence Number
 - Acceptance of Data, etc.
- A progress-state label is any label-name with prefix **progress**
- With multiple progress states, passing either one implies progress

EXAMPLE BINARY SEMAPHORE

```
proctype dijkstra()  
{  
end:    do  
      :: sema!p ->  
progress:    sema?v  
      od  
}
```



CORRECTNESS REQUIREMENTS – VALIDATION LABELS (3)

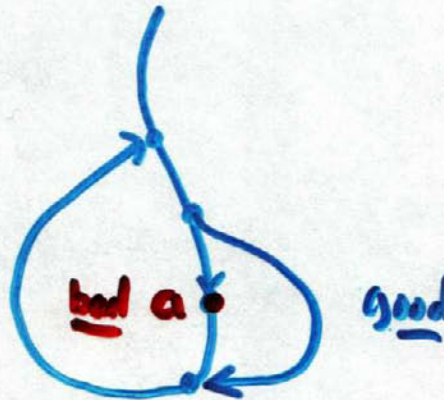
— ACCEPTANCE STATE LABELS —

- Acceptance States *may not* be part of infinite cycles
- An acceptance-state label is any label-name starting with **accept**
- If more acceptance states are defined, they all carry the same weight

EXAMPLE BINARY SEMAPHORE

```
proctype dijkstra()  
{  
end:    do  
        :: sema!p ->  
accept:      sema?v  
        od  
}
```

- Correctness Claim: It is impossible to cycle through a series of **p** and **v** operations infinitely often.
- (The claim is false for all correct implementations of the semaphore)



TEMPORAL CLAIMS

1. Are Used to Formalize LTL Formulae

LTL = Linear Time Temporal Logic

2. Mechanics

LTL $\rightarrow$ Büchi Automata $\rightarrow$ Never Claim

3. Original State Space (OSS) =

Asynchronous Product of Processes $P_1 \times P_2 \times \dots$

4. Extended State Space (ESS) =

Synchronous Product of OSS $\times$ NC

(Tight Mapping of Propositions $\rightarrow$ Behavior)

5. Acceptance Cycles in ESS are Never Claim Violations

6. Simple Example ($[] p$)

```
never
{
    do
        :: p
        :: !p  $\rightarrow$  break
    od;
accept: do
    :: skip
od
}
```

VALIDATION (LINEAR TIME TEMPORAL LOGIC FORMULAE)

- “Three classes of temporal logic formulae covering the majority of properties one would ever wish to verify”
- “An invariance property refers to an assertion p , and requires that p is an invariant over all the computations of a program P , i.e., all the states arising in a computation of P satisfy p .”

```
never {          /* formula:  $\Box p$  */
  do
    ::  $p$ 
    ::  $!p \rightarrow \text{break}$ 
  od
}
```

- “A response property refers to two assertions p and q , and requires that every p -state (a state satisfying p) arising in a computation is eventually followed by a q -state.”

```
never {          /* formula:  $\Box (p \rightarrow \langle \rangle q)$  */
  do
    :: skip
    ::  $p \rightarrow \text{break}$ 
  od;
accept: do
  ::  $!q$ 
  od
}
```

- “A precedence property refers to three assertions p , q , and r . It requires that any p -state initiates a q -interval (i.e., an interval all of whose states satisfy q) which, either runs to the end of the computation, or is terminated by an r -state.”

```
never {          /* formula:  $\Box (p \rightarrow (q \cup r))$  */
  do
    :: skip
    ::  $p \rightarrow \text{break}$ 
  od;
  do
    ::  $q$ 
    ::  $!q \ \&\& \ !r \rightarrow \text{break}$ 
  od
}
```

CORRECTNESS REQUIREMENTS – REMOTE REFERENCING

Example

```
never {
S0:      do
          :: len(receiver) == 0
          :: receiver?[msg0] -> goto accept0
          :: receiver?[msg1] -> goto accept1
        od;
accept0:  do
          :: !Receiver[2]:P0
        od;
accept1:  do
          :: !Receiver[2]:P1
        od;
S1:      skip      /* end state (unreachable) */
}
```

- The claim remains in **S0** until **receiver** is non-empty
- On message **msg0** it will change state to **accept0**
- **Receiver[2]:P0** is a proposition that is true iff the process with **pid 2** is of type **Receiver** and at label **P0**
- The claim can only remain in **accept0** if the receiver process *never* passes its state labeled **P0**
- **Receiver[2].any** is the current value of local variable **any** in the same process. For instance:

```
assert (Receiver[2].any < 0)
```
- The **pid** of a process is returned by the **run** operator, e.g.,

```
pid = run(Receiver)
```

AUTOMATED VALIDATION V.

SIMULATION 1.

Random

Fixed

Guided ←

VALIDATION 2.

Exhaustive

Supertrace

LYNCH'S PROTOCOL 3.

AUTOMATED PROOFS

''A formal proof is one which is sufficiently detailed and carried out in a sufficiently precise formal system, so that it can be checked by a computer.''

Leslie Lamport

''At some point you have to execute the program to perform the proof.''

Susan Owicki

THE STATE OF THE ART ?

● AN ANECDOTE — Mutual Exclusion

```
byte in, x, y, z;          /* shared globals, initially 0 */

proctype user(byte me)
{
L1: x = me;
L2: if
    :: (y != 0 && y != me) -> goto L1/* try again */
    :: (y == 0 || y == me) -> goto L3/* proceed */
    fi;
L3: z = me;
L4: if
    :: (x != me) -> goto L1/* try again */
    :: (x == me) -> goto L5/* proceed */
    fi;
L5: y = me;
L6: if
    :: (z != me) -> goto L1/* try again */
    :: (z == me) -> goto L7/* proceed */
    fi;
L7:          /* success */
    in = in+1; /* count competitors */
    assert(in == 1); /* access resource */
    in = in - 1;
    goto L1    /* repeat */
}
```

- 223 Reachable States
- Proof of Incorrectness: 5 msec (vax 8550)

OVERVIEW

BEHAVIOR
SPECIFICATION

+

CORRECTNESS
REQUIREMENTS

->

AUTOMATED
VALIDATION

● PROCESSES

● ASSERTIONS

● SIMULATION

● CHANNELS

● VALIDATION LABELS

● EXHAUSTIVE

● VARIABLES

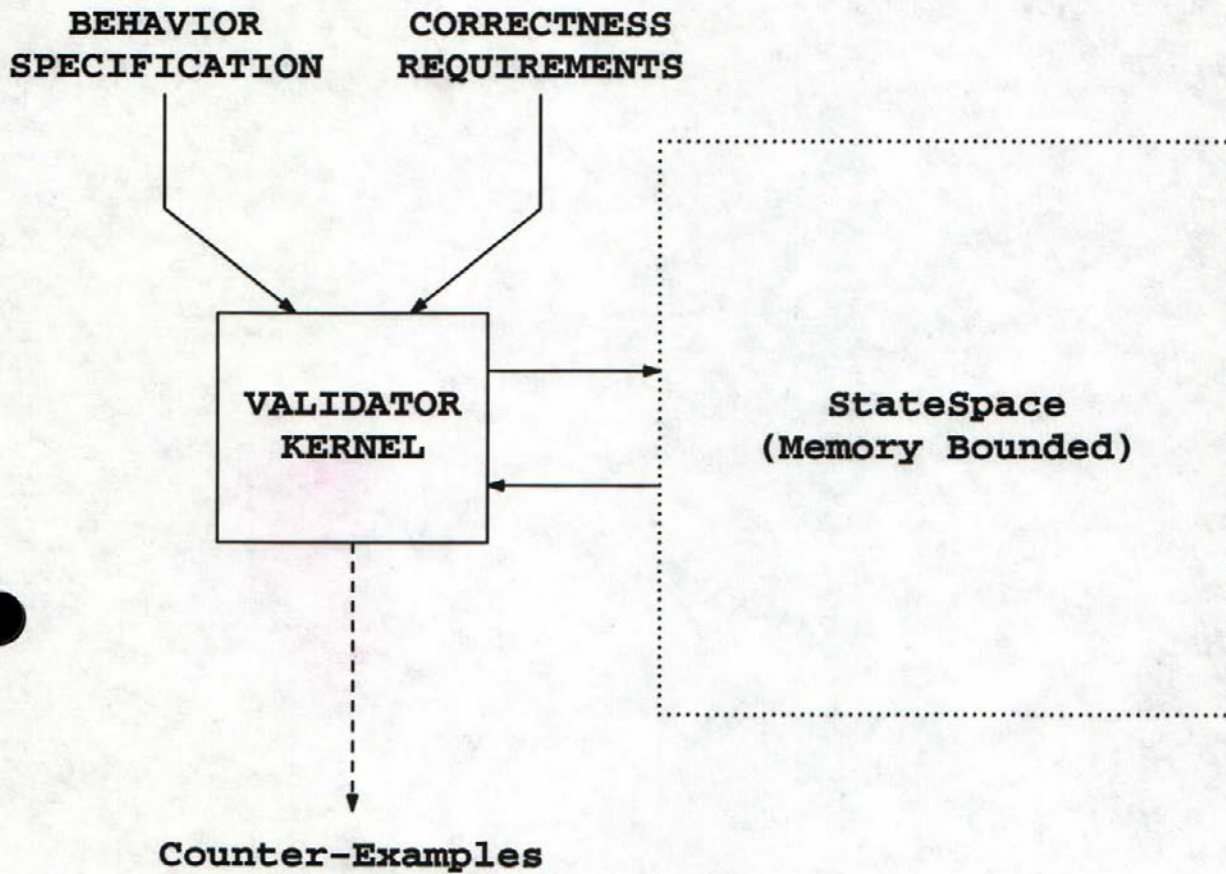
● TEMPORAL LOGIC

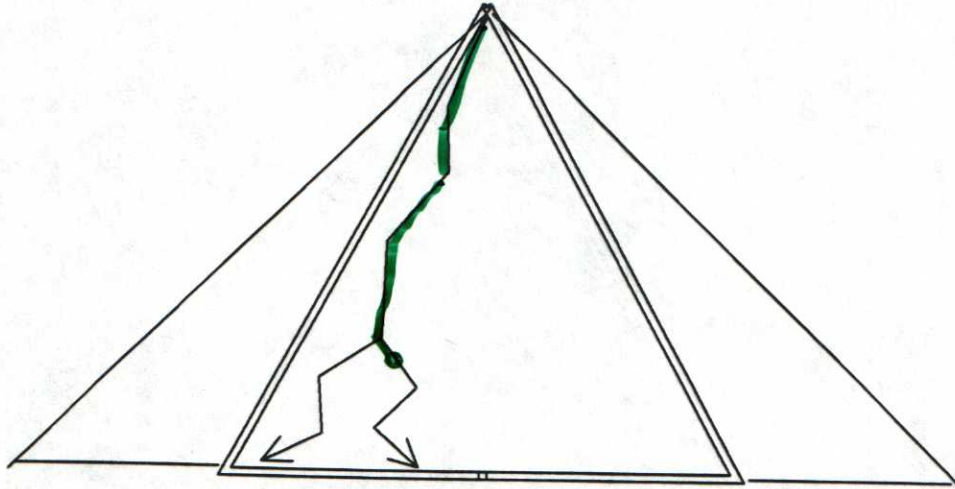
● BITSTATE

COMPUTATIONAL COMPLEXITY

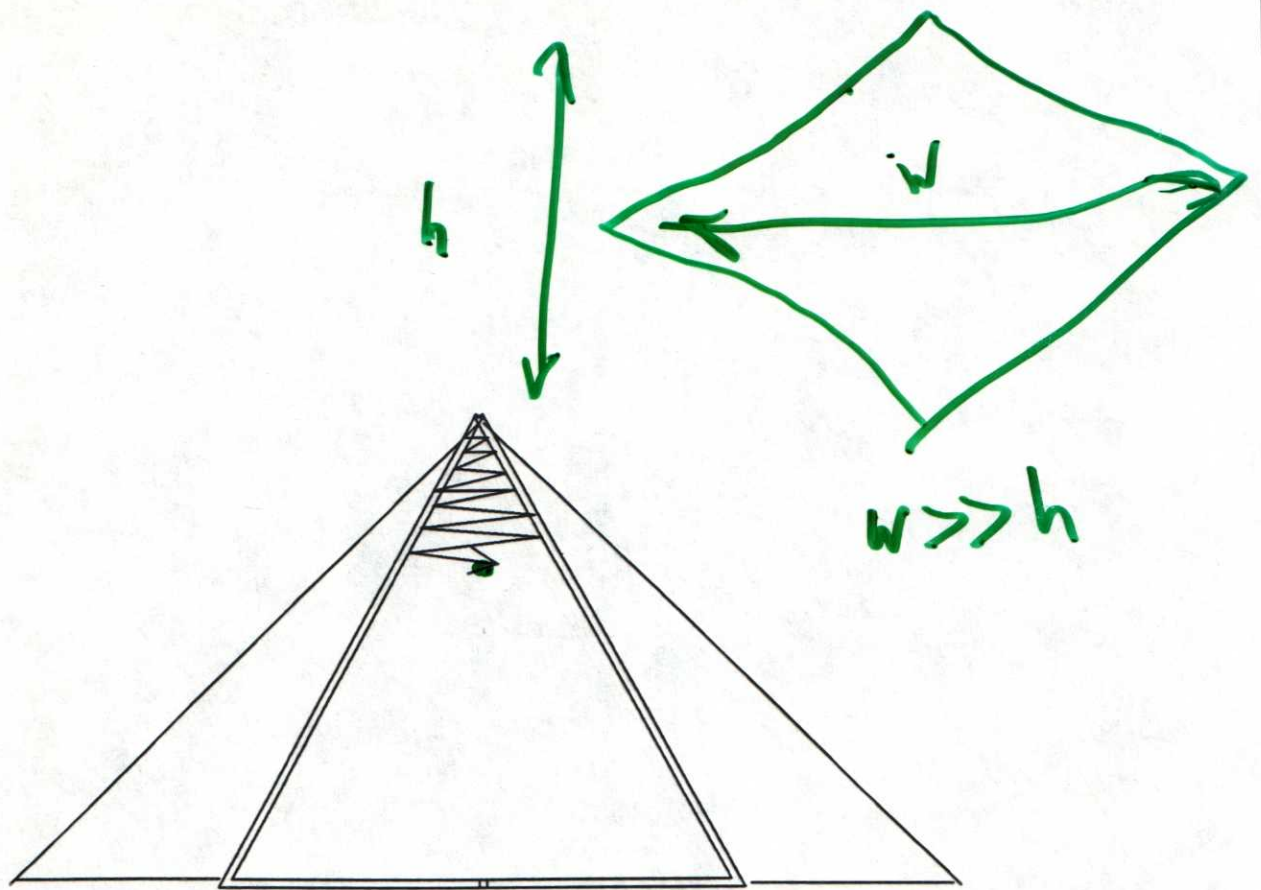
- Reachability Analysis is PSPACE Complete
- Min. Expense is Linear in the Number of Reachable States R
- All State Properties incur cost R
 - Assertion Violations
 - Unspecified Reception
 - Improper Termination
 - Absence of Deadlock
 - Buffer Overflow
- Non-Progress Cycles incur cost $2 \times R$
- Acceptance Cycles incur cost $2 \times R$
- Temporal Claims of N states incur cost $N \times R$
- Temporal Claims + Acceptance Cycles cost $2 \times N \times R$

AUTOMATED VALIDATION

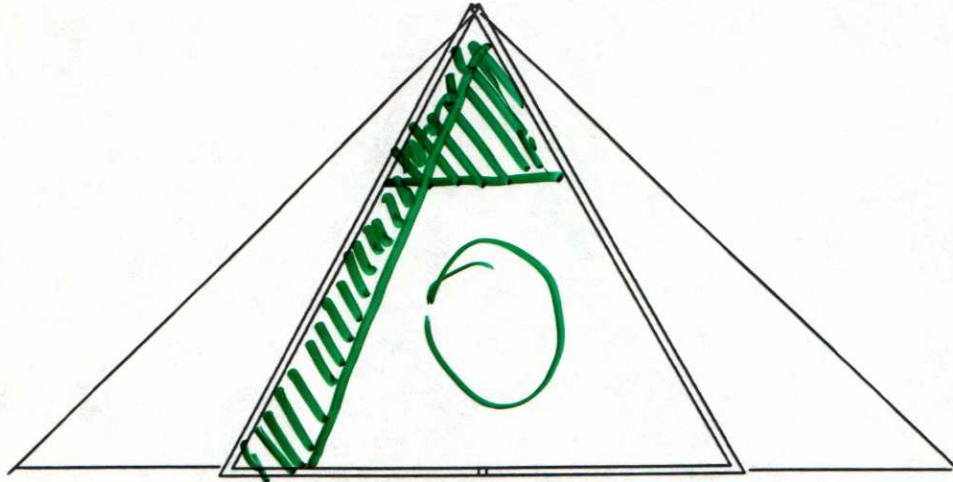




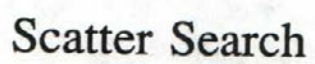
Depth-First Search



Breadth-First Search



Reachability Tree



Scatter Search

CRITIQUE OF THE FULL SEARCH

- **A** = States Analyzed
R = Reachable States
M = Memory Available
S = Size of One State

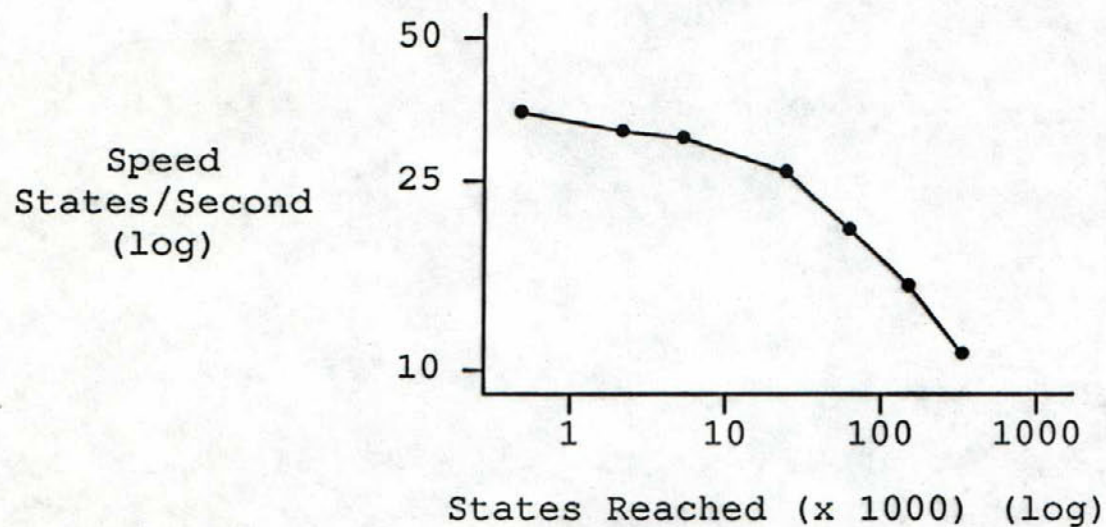
Maximum Fraction of State Space Covered:

1 only when $R \times S \leq M$

$$\frac{A}{R} = \frac{M}{R \times S} \quad \text{when } R \times S > M$$

- For Large-Scale Problems Full Search Performs Poorly

PERFORMANCE TRADITIONAL EXHAUSTIVE SEARCH



• To reduce the number of Hash Conflicts,
we must increase **H** substantially

COMPLEXITY MANAGEMENT

- Sample Problem Constraints

Memory $\leq$ 64 Mbytes
Disk $\leq$ 1 Gbytes
Speed $\sim$ 1,000 States/second
State $\geq$ 10 Kbytes
Size $\sim$ 10^9 Reachable States

- Traditional / 'Exhaustive' Reachability Analysis

$$64 \cdot \frac{10^6}{10^4} : 10^9 \rightarrow 0.001 \% \text{ Coverage per run}$$

in $10^6 - 10^7$ seconds (1 - 10 days)

Using Disk - 100 times slower

$$10^3 \cdot \frac{10^6}{10^4} : 10^9 \rightarrow 0.01 \% \text{ Coverage per run}$$

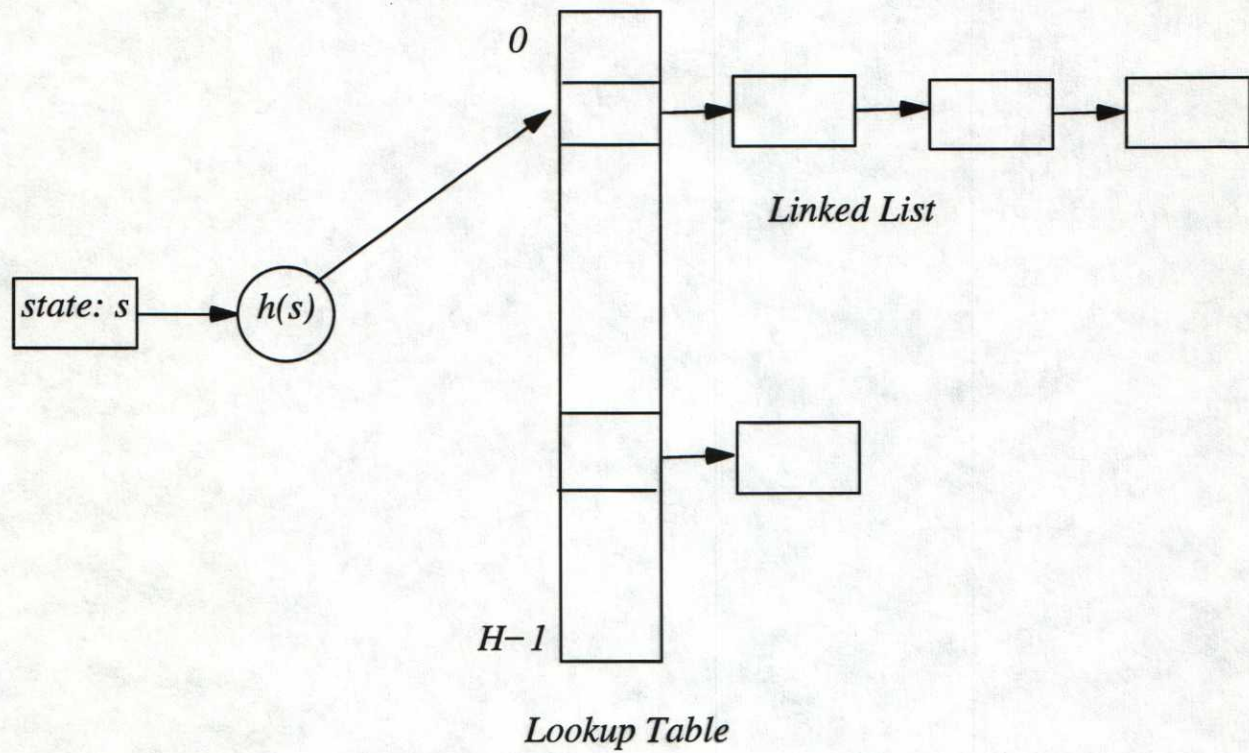
in $10^7 - 10^8$ seconds (10 - 100 days)

- Bitstate / Supertrace Reachability Analysis

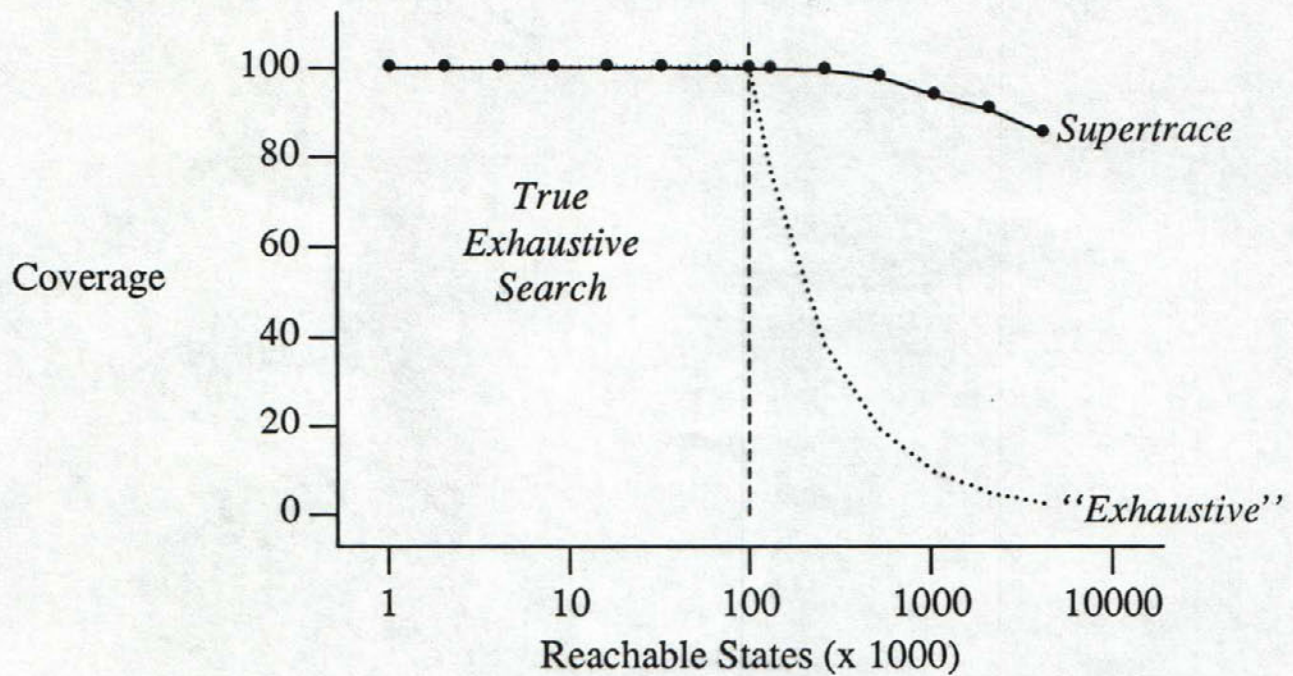
$$8.64 \cdot 10^6 : 10^9 \rightarrow 50 \% \text{ Coverage per run}$$

in $10^4 - 10^5$ seconds (3 hours - 1 day)

STATE SPACE MAINTENANCE – Hash Table Lookup

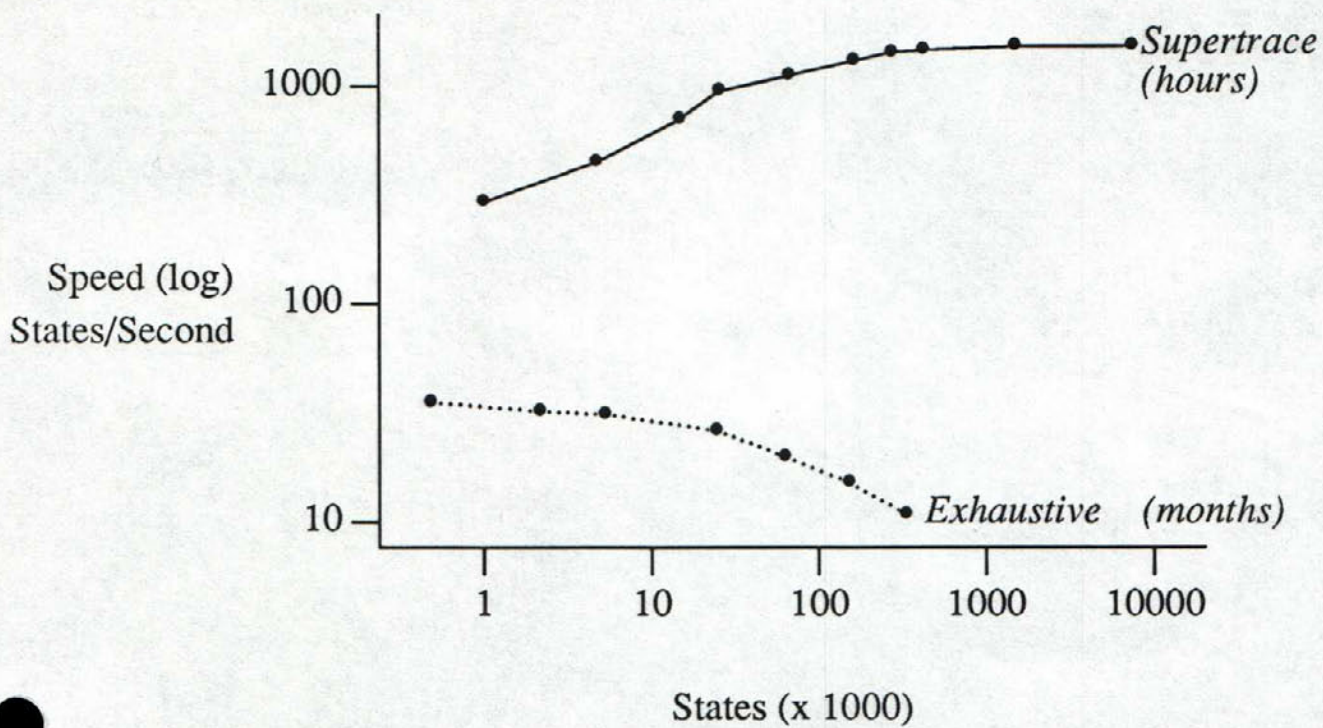


COMPARISON OF RELATIVE COVERAGE (%)



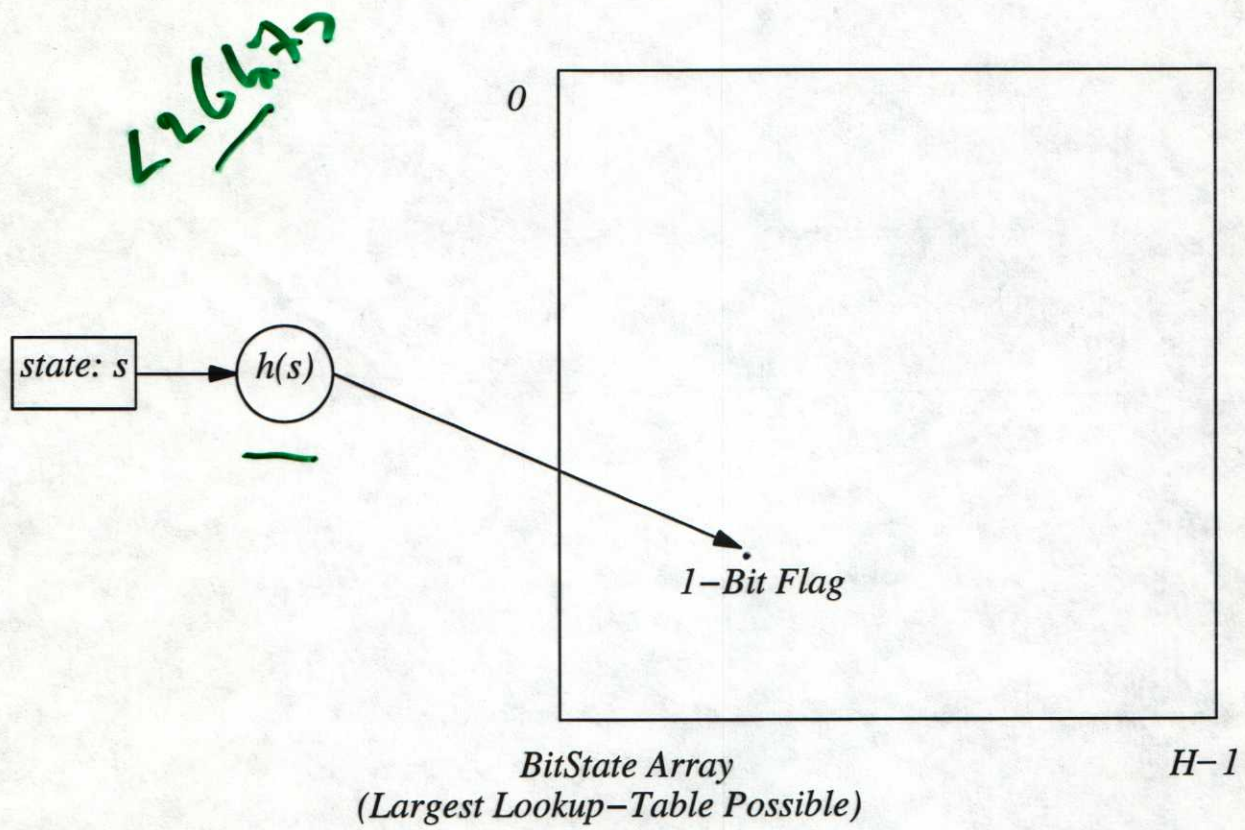
- 'Exhaustive' Searches turn into low-quality Partial Searches

PERFORMANCE: EFFECT OF HASH CONFLICTS



- Exhaustive searches slow down under Overload

STATE SPACE MAINTENANCE – SUPERTRACE



USING "SPIN"

VALIDATION (SUPERTRACE)

- Problem, assuming an exhaustive run is impossible:
Can you estimate the State Space Size from *only* a Supertrace run?

- Define "hash factor" to be

$$Hf = \frac{\text{No of Bits Available}}{\text{No of States Analyzed}}$$

- For supertrace search mode: $Hf \geq 2$
For exhaustive search mode: $Hf \geq 8 \times \text{Sizeof(State)}$

Correlation between Hash Factor and Coverage

Search	Hf	Nr.States	Collisions	Memory	Coverage
exhaustive	—	334,151	66,455	45.6 Mb	100%
supertrace	100.9	332,316	1,835	9.9 Mb	99.45%
supertrace	50.9	329,570	4,581	7.9 Mb	98.62%
supertrace	25.7	326,310	7,841	6.9 Mb	97.65%
supertrace	13.0	322,491	11,660	6.3 Mb	96.51%

LYNCH's TWO REQUIREMENTS (line 32-33)

32 This protocol has the property that
33 every message fetched by A is
 received error-free at least once
 and accepted at most once by B.

1. Every message Fetched by A

 is Accepted at least once by B

-> Define Progress as Acceptance of message by B
 Prove Absence of Non-Progress Cycles

2. No message is Accepted more than once

-> Place an Assertion at acceptance point in B
 `assert(mr == (last_mr+1)%MAX)`
 Prove Absence of Assertion Violations

VALIDATION MODEL OF SENDER

```
1  proctype A(chan in, out)
2  {   byte mt;           /* message data          */
3     bit  at;           /* alternation bit       */
4     bit  vr;           /* verify bit            */
5
6     FETCH;             /* get a new mesg        */
7     out!data(mt, at);   /* send it                */
8     do
9       :: in?control(vr) -> /* line 11, await response */
10        if
11          :: (vr == 1) -> /* line 12, correct send */
12            FETCH;       /* line 17, new message    */
13            at = 1-at;    /* line 18, toggle bit     */
14          :: (vr == 0) -> /* line 13, send error     */
15            skip         /* line 16, don't fetch    */
16        fi;
17        out!data(mt, at) /* line 16                */
18      :: in?error(vr) -> /* line 14-15, recv error */
19        out!data(mt, at) /* line 16                */
20    od
21 }
```

To send a sequence of integers in the data field:

```
#define FETCH      mt = (mt+1)%MAX
```

VALIDATION MODEL OF RECEIVER

```
1  proctype B(chan in, out)
2  {   byte mr;           /* message data          */
3     byte last_mr;       /* mr of last error-free msg */
4     bit  ar;            /* alternation bit          */
5     bit  lar;           /* ar of last error-free msg */
6
7     do
8       :: in?error(mr,ar) ->           /* lines 7-10    */
9         out!control(0)                /* lines 8,25,26 */
10      :: in?data(mr,ar) ->            /* lines 7-10    */
11        out!control(1);              /* lines 8,25,26 */
12        if                          /* line 20        */
13          :: (ar == lar) ->           /* line 21        */
14            skip                     /* line 23        */
15          :: (ar != lar) ->          /* line 24        */
16            ACCEPT;                  /* line 24        */
17            lar = ar;                /* line 22        */
18            last_mr = mr
19        fi
20    od
21 }
```

duplicate:

progress:

The same message cannot be accepted twice in a row,
(line 33 from the informal specification):

```
#define ACCEPT  assert(mr == (last_mr+1)%MAX)
```

EXAMPLE: NON-IDEAL TRANSMISSION CHANNEL

(Distortion and Loss, No Insertion or Reordering)

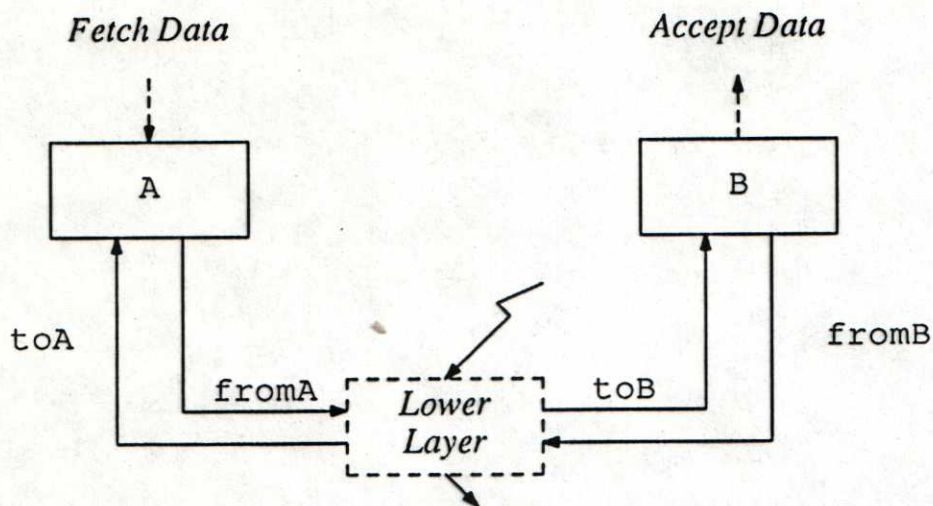


Figure 2 — A-B protocol

(Boxes are Processes — Arrows are Channels)

```
0  mtype = { data, control, error }
1  proctype lower(chan fromA, toA, fromB, toB)
2  {   byte d; bit b;
3
4      do
5          :: fromA?data(d,b) ->
6              if
7                  :: toB!data(d,b)           /* correct */
8                  :: toB!error                /* distorted */
9                  :: skip                     /* lost mesg */
10             fi
11          :: fromB?control(b) ->
12              if
13                  :: toA!control(b)
14                  :: toA!error
15                  :: skip
16              fi
17      od
18 }
```

USING "SPIN"

SIMULATION (FIXED SEED + debugging + printf's) (4)

- For instance, expand the ACCEPT macro:

```
#define ACCEPT    printf("ACCEPT %d\n",mr); \
                  assert (mr==(last_mr+1)%MAX)
```

- New output is:

```
pipe: spin -n123 -r lynch0 | grep -v lower
proc  2 (B)      line   8, Recv error,0,0 <- queue 3 (in)
proc  1 (A)      line   9, Recv control,0 <- queue 1 (in)
proc  2 (B)      line  10, Recv data,1,0  <- queue 3 (in)
proc  1 (A)      line   9, Recv control,1 <- queue 1 (in)
proc  2 (B)      line  10, Recv data,2,1  <- queue 3 (in)
ACCEPT 2
spin: "./lynch0.B" line 16: assertion violated
#processes: 4

      _p = 9
proc  2 (B)      line 16 (state 9)
proc  1 (A)      line  8 (state 14)
proc  0 (_init) line 23 (state 5)
4 processes created
```

- Error is now visible.

- The first message accepted is 2 not 1.
- Message 1 is received correctly but not accepted.

- Alternation bit on first message is set incorrectly

- Line 13 proctype B

USING "SPIN"

SIMULATION (FIXED SEED - Corrected Protocol) (5)

- Giving lar a non-zero initial value solves the problem:

```
1  proctype B(chan in, out)
2  {    byte mr;          /* message data          */
3      byte last_mr;      /* mr of last error-free msg */
4      bit  ar;           /* alternation bit          */
>5      bit lar=1;        /* ar of last error-free msg */
6
7      do
8          :: in?error(mr,ar) ->          /* lines 7-10      */
9              out!control(0)             /* lines 8,25,26   */
10         :: in?data(mr,ar) ->           /* lines 7-10      */
11             out!control(1);           /* lines 8,25,26   */
12             if                        /* line 20          */
13                 :: (ar == lar) ->      /* line 21          */
14                     skip               /* line 23          */
15                 :: (ar != lar) ->      /* line 24          */
16                     ACCEPT;            /* line 24          */
17                     lar = ar;          /* line 22          */
18                     last_mr = mr
19             fi
20         od
21 }
```

- Problem solved (or is it?):

```
$ spin -m123 lynch0
ACCEPT 1
ACCEPT 2
ACCEPT 3
ACCEPT 4
ACCEPT 5
ACCEPT 6
ACCEPT 7
ACCEPT 0
ACCEPT 1
ACCEPT 2
ACCEPT 3
...
```

USING "SPIN"

VALIDATION (EXHAUSTIVE) (1)

- Simulation alone cannot *prove* correctness
- Exhaustive validation, is performed in three steps:

1. *Generate* the analyzer source
2. *Compile* this source
3. *Run* the executable analyzer.

- (Removed ACCEPT printf debugging statements)

```
$ spin -a lynch0          # STEP 1: generate analyzer
$ cc -o pan pan.c         # STEP 2: compile sources
$ time pan                # STEP 3: run
full statespace search for:
    assertion violations and invalid endstates
vector 84 byte, depth reached 350, errors: 0
    1558 states, stored
    2 states, linked
    1169 states, matched          total:      2729
hash conflicts: 201 (resolved)
(max size 2^18 states, stackframes: 0/102)
```

```
...
0.4u 0.3s 5r      pan
```

- Longest Sequence is 350 statements (depth reached),
- There are 1558 unique Reachable System States
- Each State occupies 84 bytes of storage.

BACK TO LYNCH's PROTOCOL

Completing the Validation Model

- The informal specification says

27 Initialization of this scheme depends upon A and B agreeing on an
28 initial setting of the alternation bit. This is accomplished by
29 an A to B message whose error-free reception (but not necessarily
30 acceptance) forces B's setting of the alternation bit. Multiple
31 receptions of such a message cannot do harm.

- One way to formalize this is:

```
1 proctype A(chan in, out)
2 {   byte mt;           /* message data   */
3     bit  at;           /* alternation bit */
4     bit  vr;           /* verify bit     */
5
> 6     out!data(0,1);    /* initialize     */
> 7     do
> 8         :: in?error(vr) -> /* ack error     */
> 9             out!data(0,1) /* repeat        */
>10         :: in?control(0) -> /* send error    */
>11             out!data(0,1) /* repeat        */
>12         :: in?control(1) -> /* success: done */
>13             break
>14     od;
15     FETCH;
    ...
30 }
```

USING "SPIN"

VALIDATION (EXHAUSTIVE - Completed Protocol Model) (2)

- Exhaustive validation succeeds, independent of initial value of lar.

```
1 proctype B(chan in, out)
2 (   byte mr;           /* message data           */
3   byte last_mr;       /* mr of last error-free msg */
4   bit  ar;            /* alternation bit           */
5   bit  lar=1;         /* ar of last error-free msg */
> 6   bit ini;           /* lines 27-31              */
7
8   do
9     :: in?error(mr,ar) -> /* lines 7-10              */
10      out!control(0)      /* lines 8,25,26            */
11     :: in?data(mr,ar) -> /* lines 7-10              */
12      out!control(1);     /* lines 8,25,26            */
>13     if /* lines 27-31 */
>14     :: (!ini) -> /* lines 27-31 */
>15         ini = 1; /* lines 27-31 */
>16         lar = ar /* lines 27-31 */
>17     :: ini -> /* lines 27-31 */
18         if /* line 20 */
19         :: (ar == lar) -> /* line 21 */
20             skip /* line 23 */
21         :: (ar != lar) -> /* line 24 */
22             ACCEPT; /* line 24 */
23             lar = ar; /* line 22 */
24             last_mr = mr
25         fi
26     fi /* lines 27-31 */
27   od
28 )
```

WHAT HAVE WE PROVEN, SO FAR?

- The behavior is cyclic → use progress labels, acceptance labels or temporal claims. Acceptance of a correct message is *progress*

```

1 proctype B(chan in, out)
2 {   byte mr;           /* message data           */
3     byte last_mr;      /* mr of last error-free msg */
4     bit  ar;           /* alternation bit           */
5     bit  lar=1;        /* ar of last error-free msg */
6     bit  ini;          /* lines 27-31              */
7
8     do
9         :: in?error(mr,ar) ->          /* lines 7-10              */
10            out!control(0)              /* lines 8,25,26           */
11         :: in?data(mr,ar) ->          /* lines 7-10              */
12            out!control(1);            /* lines 8,25,26           */
13         if                             /* lines 27-31             */
14             :: (!ini) ->              /* lines 27-31             */
15                 ini = 1;              /* lines 27-31             */
16                 lar = ar              /* lines 27-31             */
17             :: ini ->                /* lines 27-31             */
18                 if                  /* line 20                  */
19                     :: (ar == lar) -> /* line 21                  */
20                         skip          /* line 23                  */
21                     :: (ar != lar) -> /* line 24                  */
>22 progress: ACCEPT; /* line 24                  */
23                     lar = ar; /* line 22                  */
24                     last_mr = mr
25             fi
26         fi                             /* lines 27-31             */
27     od
28 }

```

USING "SPIN"

VALIDATION (FINDING NON-PROGRESS CYCLES)

```
$ spin -a lynch00      # STEP 1: generate analyzer
$ cc -o pan pan.c      # STEP 2: compile source
$ pan -l               # STEP 3: option -l for loop analysis
pan: non-progress cycle (at depth 21)
pan: wrote pan.trail
full statespace search for:
    assertion violations and non-progress loops
search was not completed
vector 88 byte, depth reached 31, non-progress loops: 1
    65 states, stored
    2 states, linked
    17 states, matched          total:          84
hash conflicts: 0 (resolved)
(max size 2^18 states, stackframes: 0/9)

$ ls -l pan.trail
-rw-rw-rw- 1 gerard other 477 Apr 16 13:42 pan.trail
```

USING "SPIN"

SIMULATION (GUIDED)

```
$ ls -l pan.trail
-rw-rw-rw- 1 gerard other 477 Apr 16 13:42 pan.trail
```

- SPIN with option `-t` will reproduce the error-sequence.

```
$ spin -t -r lynch00
proc 3 (lower) line 5, Recv data,0,1 <- queue 2 (fromA)
proc 2 (B) line 11, Recv data,0,1 <- queue 3 (in)
proc 3 (lower) line 11, Recv control,1 <- queue 4 (fromB)
proc 1 (A) line 12, Recv control,data <- queue 1 (in)
proc 3 (lower) line 5, Recv data,1,0 <- queue 2 (fromA)
proc 2 (B) line 9, Recv error,0,1 <- queue 3 (in)
<<<<<START OF CYCLE>>>>>
proc 3 (lower) line 11, Recv control,0 <- queue 4 (fromB)
proc 1 (A) line 19, Recv control,0 <- queue 1 (in)
proc 3 (lower) line 5, Recv data,1,0 <- queue 2 (fromA)
proc 2 (B) line 9, Recv error,0,1 <- queue 3 (in)
spin: trail ends after 46 steps
step 46, #processes: 4
      _p = 2
proc 3 (lower) line 4 (state 11)
proc 2 (B) line 10 (state 2)
proc 1 (A) line 18 (state 24)
proc 0 (_init) line 23 (state 5)
4 processes created
```

- *Non-Progress Cycle:*
 - A sends message
 - C distorts it
 - B rejects it as an error
 - A retransmits the message
 - etc.

USING "SPIN"

VALIDATION (NON-PROGRESS CYCLES — REFINEMENTS)

- We're not interested in infinite repetition of distortion:

```
1  proctype lower(chan fromA, toA, fromB, toB)
2  (   byte d; bit b;
3
4      do
5          :: fromA?data(d,b) ->
6              if
7                  :: toB!data(d,b)          /* correct */
8                  ::
9  progress0:      toB!error                /* distorted */
10                 fi
11          :: fromB?control(b) ->
12              if
13                  :: toA!control(b)
14                  ::
15  progress1:      toA!error
16                 fi
17      od
18  )
```

USING "SPIN"

VALIDATION (NON-PROGRESS CYCLES - THE FINAL PROOF)

```
$ spin -a lynch01      # STEP 1: generate analyzer
$ cc -o pan pan.c      # STEP 2: compiler sources

$ pan -?               # check the options
unknown option
-cN stop at Nth error (default=1)
-l find non-progress loops
-mN max depth N (default=10k)
-wN hashtable of 2^N entries (default=18)

$ pan -l               # STEP 3: run loop analysis
full statespace search for:
    assertion violations and non-progress loops
vector 88 byte, depth reached 399, non-progress loops: 0
    4433 states, stored
    4 states, linked
    5039 states, matched          total:      9476
hash conflicts: 2107 (resolved)
(max size 2^18 states, stackframes: 0/108)
```

USING "SPIN"

VALIDATION (TEMPORAL CLAIMS)

- In the absence of message distortion, there can be no infinite series of duplicate messages, without any correct message getting through.
- Mark line 20 in proctype B with label Dup
- The illegal cycle exists only if B can pass "Dup" without passing "progress" (line 12).

```
1  never {
2  /* there is no cycle through ``Dup`` without ``progress`` */
5  accept:    do
6              ::      do      /* arbitrary prefix */
7                      :: !B[2]:Dup && !B[2]:progress
8                      :: B[2]:Dup -> break
9                      od;
10             do
11             :: B[2]:Dup
12             :: !B[2]:Dup && !B[2]:progress -> break
13             od
14             od
15 }
```

● It Cannot Happen

```
$ spin -a lynch02      # STEP 1: generate the analyzer
$ cc -o pan pan.c      # STEP 2: compile the sources
$ pan                  # STEP 3: run analysis
full statespace search on behavior restricted to claim for:
    assertion violations
    and absence of acceptance labels in all cycles
vector 88 byte, depth reached 81, errors: 0
    263 states, stored
    2 states, linked
    196 states, matched          total:          461
hash conflicts: 76 (resolved)
(max size 2^18 states, stackframes: 0/13)
```

RULES OF SOUND (PROTOCOL) DESIGN

1. VALIDATE TWICE, IMPLEMENT ONCE
2. EXPECT THE UNEXPECTED
3. GOOD DESIGNS ARE SPEED INDEPENDENT
4. DESIGN ERRORS ARE WHERE YOU ASSUME THEY ARE NOT
5. ANY GOOD PROOF CAN BE AUTOMATED
6. IF YOU CAN'T PROVE IT, YOU CAN'T TRUST IT
7. DESIGN BY STEPWISE REFINEMENT
8. PROVE BY STEPWISE ABSTRACTION
9. BUILD A PROTOTYPE
10. DO IT