UTA

SPIN Overview

- Introduction
- SPIN by Examples
- The PROMELA Language
- Defining Correctness Claims
- Summary

UTA

SPIN is ...

... a tool that is designed to verify correctness requirements for multi-threaded and distributed software systems.

The SPIN project is developed by Gerard Holzmann in Bell Lab. Additional information can be found at http://spinroot.com.

Acknowledgement: The lecture notes use some materials from Dr. Holzmann's lectures at Caltech.

Model Checking (1)

The theoretic foundation of SPIN is model checking, i.e., an automatic technique to verify the correctness of finite state systems.

It involves checking a desired property over a system (or a model of the system) through exhaustive search of all the reachable states.

Important: Deductive verification can be used to verify systems with infinite state space.

Model Checking (2)

In model checking, a system is usually modeled as a state transition graph, and a desired property is usually specified in a temporal logic formula.

The verification process is to check if the model (i.e. the state transition graph) satisfies the specified temporal logic formula.

Correctness

A system is correct if it meets its requirements:

- Correctness cannot be proved in any absolute sense. What we can prove is that a system does or does not satisfy certain properties.
- It is human judgment to conclude whether satisfying these properties implies 'correctness'.
- Getting the properties right is as important as getting the system right.

Requirements

Some requirements are universal, such as freedom from deadlock, livelock, and starvation.

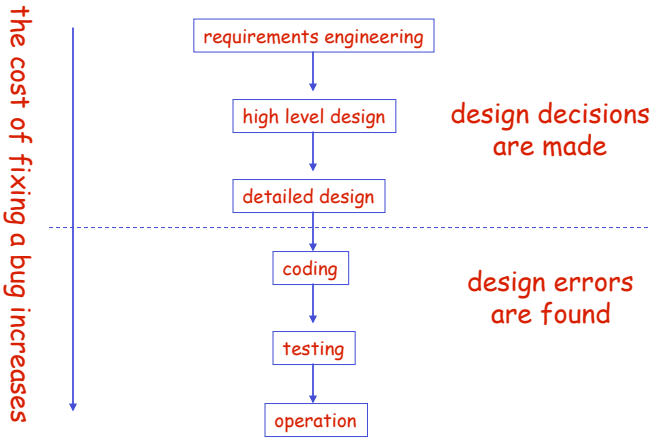
There are also many application-dependent requirements, such as proper termination states, reliable data transfer, and so on.

The SPIN tool can be used to check both types of requirements.

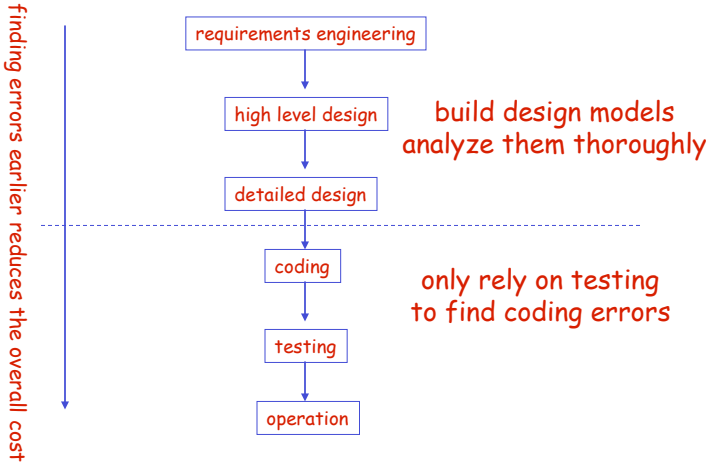
Model

- A model is an abstraction of reality
  - it should be less detailed than the artifact being modeled
  - the level of detail is selected based on its relevance to the correctness properties
  - the objective is to gain analytical power by reducing details
- A model is a design aid
  - it often goes through different versions, describing different aspects of reality, and can slowly become more accurate, without becoming more detailed.

The Philosophy (1)



The Philosophy (2)



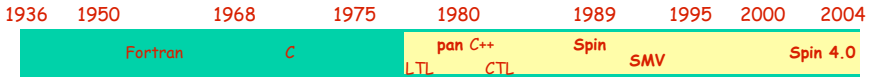
In practice ...

There are two ways of working with SPIN:

- Use SPIN to verify a design model before implementation.
- Extract a model from an implementation and then verify it using SPIN.

UTA

A bit history



1936: first theory on computability, e.g., Turing machines

1940-50: the first computers are built

1955: early work on tense logics (predecessors of LTL)

1960: early work on w-automata theory, e.g., by J.R. Buchi

1968: two terms introduced: software crisis, software engineering

1975: Edsger Dijkstra's paper on Guarded Command Language
1978: Tony Hoare's paper on Communicating Sequential Processes

1977: Amir Pnueli introduces LTL for system verification

1986: Pierre Wolper and Moshe Vardi define the automata theoretic framework for LTL model checking

1986: Mazuklewicz paper on trace theory

1989: Spin V0: verification of w-regular properties

1993: SMV Model Checker (Ken McMillan)

1995: Partial order reduction in SPIN, LTL conversion in SPIN (Doron Peled)

the two most popular logic model checkers:
Spin: an explicit state LTL model checker
based on automata theoretic verification
targeting software verification (asynchronous systems)
SMV: a symbolic CTL model checker
targeting hardware circuit verification (synchronous systems)

UTA

What we will learn ...

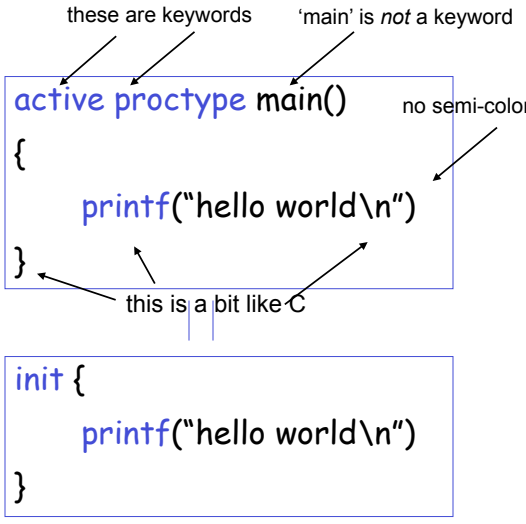
... is mainly centered around the following questions (1) how to build system models; (2) how to specify logic properties; and (3) how to perform the analysis.

We will not only learn how to use the SPIN tool, but also the foundation upon which the tool is built.

Overview of SPIN

- Introduction
- SPIN by Examples
- The PROMELA Language
- Defining Correctness Claims
- Summary

Hello World



a simulation run:

```
$ spin hello.pml
hello world
1 process created
$
```

a verification run:

```
$ spin -a hello.pml
$ gcc -o pan pan.c
$ ./pan
... depth reached 2, errors: 0
$
```

UTA

Producer/Consumer

```
mtype = { P, C };
mtype  turn = P;
active proctype producer ()
{
    do
    :: (turn == P) ->
         printf ("Produce\n");
         turn = C
    od
}
active proctype consumer ()
{
    do
    :: (turn == C) ->
          printf ("Consume\n");
          turn = P
    od
}
```


UTA

Loop and Selection

```
active proctype producer ()
{
    do
    :: (turn == P) ->
          printf ("Produce\n");
          turn = C
    od
}
```

—

```
active proctype producer ()
{
again:  if
        :: (turn == P) ->
              printf ("Produce\n");
              turn = C
        fi
        goto again
}
```

UTA

Guard Conditions

A guard condition in a loop or selection structure determines whether or not the statement that follows can be selected for execution.

If no guard condition is true, then the process blocks. If more than one guard condition is true, then the selection is non-deterministic.

The else guard condition is true iff all other guard conditions are false.

```
(turn == P) ->
```

=

```
wait: if
      :: (turn == P) ->
      :: else -> goto wait
      fi;
```


UTA

Alternation

If there is only one producer and one consumer, then the earlier program guarantees a strict alternation between producer and consumer.

Now, what if there are multiple producers and consumers?

UTA

A revised version

```
mtype = { P, C, N }
mtype turn = P;
pid who;

inline request (x, y, z) {
  atomic { x == y -> x = z; who = _pid; }
}

inline release (x, y) {
  atomic { x = y; who = 0}
}

active [2] proctype producer ()
{
  do
  :: request ( turn, P, N ) ->
       printf ("P%d\n", _pid);
       assert (who == _pid);
       release ( turn, C)
  od
}

active [2] proctype consumer ()
{
  do
  :: request ( turn, C, N) ->
       printf ("C%d\n", _pid);
       assert (who == _pid);
       release (turn, P)
  od
}
```


UTA

Critical Section

Recall that there are two requirements for the critical section problem:

- Mutual exclusion – At any given time, there is at most one process inside the CS.
- Eventual entry – A process that requests to enter the CS will be able to enter eventually.

Peterson's Algorithm

```
bool turn, flag[2];
byte cnt;

active [2] proctype P ()
{
   pid i, j;
   i = _pid;
   j = 1 - _pid;

again:
   flag[i] = true;
   turn = i;
   (flag[j] == false || turn != i) -> /* wait until true */
   cnt ++;
   assert (cnt == 1);
   cnt --;
   flag[i] = false;
   goto again;
}
```


Executability

In PROMELA, every type of statement can be used as a guard in any context.

A statement is executable if and only if the expression evaluates to true. Otherwise, it blocks the process, until it becomes executable.

Important: All statements are side-effect free when they evaluate to false.

UTA

A data exchange protocol

```
mtype = { ini, ack, dreq, data, shutup, quiet, dead }
chan M = [1] of { mtype };
chan W = [1] of { mtype };

active proctype Mproc ()
{
   W!ini;
   M?ack;
   if
   :: W!shutup
   :: W!dreq;
        M?data ->
        do
        :: W!data
        :: W!shutup;
             break
        od
   fi
   M?shutup;
   W!quiet;
   M?dead
}
active proctype Wproc ()
{
   W?ini;
   M!ack;
   do
   :: W?dreq -> M!data
   :: W?data -> skip
   :: W?shutup -> M!shutup; break
   od;
   W?quiet;
   M!dead
}
```

channel declaration

send a message

receive a message

UTA

SPIN Overview

- Introduction
- SPIN by Examples
- The PROMELA Language
- Defining Correctness Claims
- Summary

UTA

A Meta Language

PROMELA stands for PROcess MEta LAnguage. It is designed to facilitate the description of design abstractions.

In particular, the language is targeted to the description of concurrent software systems. The emphasis is on the synchronization and communication aspects.

UTA

Verification Model

PROMELA is used to describe a verification model, not an implementation

- A verification model is an abstraction of a design, which omits many low level details that would exist in an implementation.
- On the other hand, such a model often includes the behavior of the environment and a specification of correctness properties, which are usually not included in an implementation.

Building Blocks

A PROMELA model is typically built from asynchronous processes, message channels, synchronization statements, and structured data.

Deliberately, a model often has few computations. In addition, it does not have the notion of time or clock.

Important: What is the difference between concurrent systems and real-time systems?

Process Instantiation (1)

```
active [2] proctype MyProc ()
{
  printf ("my pid is: %d\n", _pid);
}
```

```
proctype MyProc (byte x)
{
  printf ("x = %d, pid = %d\n", x, _pid);
}
init {
  run MyProc (0);
  run MyProc (1);
}
```

Process Instantiation (2)

Question: What is the difference between the two ways for process instantiation?

Process Instantiation (3)

The value of a run expression evaluates to zero if it fails to instantiate a process; otherwise, it evaluates to the pid of the newly created process.

```
init {
  pid p0, p1;
  p0 = run MyProcess(0);
  p1 = run MyProcess(1);
  printf ("pids: %d and %d\n", p0, p1);
}
```

Process Termination and Death (1)

A process "terminates" when it reaches the end of its code; a process can only "die" if all child processes have died.

When a process has terminated, it can no longer execute statements, but is still maintained in the system. It is removed from the system only after it dies.

Process Termination and Death (2)

Processes can terminate in any order, but they can only die in the reverse order.

```
active proctype splurge (int n)
{
  pid p;
  printf ("%d\n", n);
  p = run splurge (n + 1);
}
```

Basic Data Types

Type	Typical Range
bit	0, 1
bool	false, true
byte	0 .. 255
chan	1 .. 255
mtype	1 .. 255
pid	0 .. 255
short	-2^{15} .. $2^{15} - 1$
int	-2^{15} .. $2^{15} - 1$
unsigned	0 .. $2^{n} - 1$

Example Declarations

```
bit x, y;
bool turn = true;
byte a[12];
chan m;
mtype n;
short b[4] = 89;
int cnt = 67;
unsigned v : 5;
unsigned w : 3 = 5;
```

UTA

Enumeration Type

```
mtype = { apple, pear, orange, banana };
mtype = { fruit, vegetables, cardboard };

init {
  mtype n = pear;

  printf ("the value of n is ");
  printm (n);
  printf ("\n");
}
```


UTA

User-defined Type

```
typedef Field {
  short f = 3;
  byte g
};
typedef Record {
  byte a[3];
  int fld1;
  Field fld2;
  chan p[3];
  bit b
};
proctype me (Field z) {
  z.g = 12
}
init {
  Record goo;
  Field foo;

  run me(foo);
}
```

Multidimensional Array

PROMELA supports only one-dimensional array as first class objects. Multi-dimensional arrays have to declared indirectly.

```
typedef Array {
  byte el[4];
}
Array a[4];
```


Variable Scopes

There are only two levels of scope in PROMELA: global and process local.

```
init {
  int x;
  {
    int y;
    printf ("x = %d, y = %d\n", x, y);
    x ++; y ++;
  }
  printf ("x = %d, y = %d\n", x, y);
}
```

Message Channel

A message channel is used to exchange data between processes.

```
chan ch = [16] of { short, byte, bool }
```

Question: There are two ways to communicate between processes in PROMELA. What is the other one?

Channel Operations (1)

By default, a send operation is executable only if the target channel is not full.

Moreover, the number of message fields in a send operation must equal the number of message fields declared for the channel.

```
qname ! expr1, expr2, expr3
        ||
qname ! expr1 (expr2, expr3)
```

Channel Operations (2)

A receive operation, without any constant parameters, is executable only if the source channel is non-empty.

```
qname ? var1, var2, var3
        ||
qname ? var1 (var2, var3)
```


Channel Operations (3)

A receive operation may take some constants as its parameters. If so, the operation is executable only if these constants are matched in the message that is to be received.

```
qname ? cons1, var2, cons2
```

```
qname ? eval(var1), var2, cons2
```

Channel Operations (4)

```
mtype = { msg0, msg1, ack0, ack1 };
chan to_sndr = [2] of { mtype };
chan to_rcvr = [2] of { mtype };
active proctype Sender ()
{
again: to_rcvr ! msg1;
       to_sndr ? ack1;
       to_rcvr ! msg0;
       to_sndr ? ack0;
       goto again
}
active proctype Receiver ()
{
again: to_rcvr ? msg1;
       to_sndr ! ack1;
       to_rcvr ? msg0;
       to_sndr ! ack0;
       goto again
}
```


Channel Operations (5)

- Test the executability of a send or receive operation

```
qname ? [ m0 ]
```

- Poll – Read a message but do not remove it

```
qname ? < eval (x), y >
```

UTA

Channel Operations (6)

- len(qname) – returns the number of msgs
- empty(qname)/nempty(qname)
- full(qname)/nfull(qname)

UTA

Rendezvous (1)

PROMELA supports both asynchronous and synchronous or rendezvous communication.

Question: What is the difference between the two communication models?

```
chan qname = [N] of { byte }
```

```
chan qname = [0] of { byte }
```

Rendezvous (2)

```
mtype = { msgtype };
chan name = [0] of { mtype, byte };
active proctype A ()
{
   name ! msgtype (124);
   name ! msgtype (121)
}
active proctype B ()
{
   byte state;
   name ? msgtype (state);
}
```


Passing Channels

```
mtype = { msgtype };
chan glob = [0] of { chan };

active proctype A ()
{
  chan loc = [0] of { mtype, byte };
  glob ! loc;
  loc ? msgtype (121)
}

active proctype B ()
{
   chan who;
   glob ? who;
   who ! msgtype (121)
}
```

UTA

Executability (1)

PROMELA has four types of statements: print statements, assignments, I/O statements, and expression statements.

A statement is either executable or blocked. The first two types of statements are always executable. The other two depends on the current system state.

Important: An expression can be used as part of another statement. In this case, the expression is "passable" only if it evaluates to true.

UTA

Executability (2)

The meaning of statement (a == b); can be explained using the following blocks:

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

```
L:
   if
   :: (a == b) -> skip
   :: else -> goto L
   fi
```

```
do
:: (a == b) -> break
:: else -> skip
od
```

Side effects

Any expression in PROMELA must be side effect free when it evaluates to false. (Why?)

Compound Statements

We have seen the basic statements of PROMELA: print, assignment, expression, and send/receive statements.

In addition, there are five types of compound statements: (1) atomic sequence; (2) deterministic steps; (3) selection; (4) repetitions; (5) escape sequences.

Atomic Sequence

The execution of an atomic sequence cannot be interrupted.

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

```
nfull (ch) -> ch ! msg
```

```
nempty (ch) -> ch ? msg
```

Important: The atomicity can be broken if one of the statements is found to be unexecutable.

Selection (1)

Selection basically represents the branch structure. An option sequence can be selected only if its first statement or its guard is executable.

If more than one guard is executable, the choice is non-deterministic. If none of the guards is executable, the process is blocked.

There is no restriction on the types of statements that can be used as a guard.

Selection (2)

```
byte count;

active proctype counter ()
{
   if
   :: count ++
   :: count --
   fi
}
```


Repetition (1)

Repetition basically represents the loop structure. One of the option sequences is selected in a way that is similar to selection.

```
active proctype counter ()
{
  do
  :: (count != 0) ->
     if
     :: count ++
     :: count –
     fi
  :: (count == 0) -> break
  od
}
```

UTA

Repetition (2)

An else option can be selected only if no other options are executable.

```
active proctype counter ()
{
  do
  :: (count != 0) ->
    if
    :: count ++
    :: count –
    :: else
    fi
  :: else -> break
  od
}
```


UTA

The AB Protocol

```
mtype = { msg, ack };
chan to_sndr = [2] of { mtype, bit };
chan to_rcvr = [2] of { mtype, bit};

active proctype Sender ()
{    bool seq_out, seq_in;
     do
     :: to_rcvr ! msg (seq_out) ->
            to_sndr ? ack (seq_in);
            if
            :: seq_in == seq_out ->
                  seq_out = 1 – seq_out;
            :: else
            fi
     od
}

active proctype Receiver ()
{    bool seq_in;
     do
     :: to_rcvr ? msg (seq_in) ->
            to_sndr ! ack (seq_in)
     :: timeout ->
            to_sndr ! ack (seq_in)
     od

}
```

timeout vs else

- Both timeout and else are predefined boolean variables; their values are set by the system.
- else is true iff no other statement is executable in the same process; timeout is true iff no other statement is executable in the same system.
- In some sense, timeout can be considered as a system-level else.

Inline Definition (1)

The invocation of an inline is replaced with the text of the body of its definition.

```
inline example (x, y) {
  y = a;
  x = b;
  assert (x)
}
init {
  int a, b;
  example (a, b);
}
```

```
init {
  int a, b;

  b = a;
  a = b;
  assert (a);
}
```

Important: There is no concept of value passing with inline's.

Inline Definition (2)

```
inline example (x) {
   int y;
   y = x;
   printf ("%d\n", y)
}

init {
   int a;
   a = 34;
   example (a);
   y = 0;
}
```


Reading Input

Usually, a verification model should be closed, i.e., it must contain all the information that is needed to verify its properties.

This means that reading inputs is typically not allowed in a PROMELA model. The only exception is that inputs can be read during simulation runs.

Reading inputs is accomplished by reading from a built-in channel STDIN, which define a single message field of type int.

UTA

Overview of SPIN

- Introduction
- SPIN by Examples
- The PROMELA Language
- Defining Correctness Claims
- Summary

UTA

Correctness claims

... are an integral part of a verification model. They specify design requirements that should be satisfied by a system.

SPIN is primarily concerned with the possibility, rather than probability, of a requirement could be violated.

Proof of correctness

... should be independent of the following assumptions:

- the relative speeds of processes
- the time it takes to execute specific instructions
- the probability of occurrence of certain events

Safety vs Liveness

In general, there are two types of correctness properties:

- Safety – Bad things should not happen.
- Liveness – Good things must eventually happen.

Types of claims

PROMELA provides the following constructs to formalize correctness properties:

- Basic assertions
- End labels
- Progress labels
- Accept labels
- Never claims
- Trace assertions

Basic Assertions

Basic assertions are always executable. If they evaluate to true, their execution has no effect. Otherwise, an error message is triggered.

An assertion statement is the only type of correctness property checked in both simulation and verification modes.

```
assert (expression)
```

UTA

End Labels (1)

By default, SPIN checks if every process reaches the end of its code upon termination. If any process does not, a deadlock is reported.

End-state labels are used to specify valid end states other than the default one. Any label that starts with the prefix end defines an end-state label.

UTA

End Labels (2)

```
mtype { p, v };
chan sema = [0] of { mtype };
active proctype Dijkstra ()
{           byte count = 1;
end:        do
            :: (count == 1) ->
                    sema ! p; count = 0
            :: (count == 0) ->
                    sema ? v; count = 1
            od
}
active [3] proctype user ()
{           if
            :: sema ? p;
               skip;
               sema ! v;
            if
}
```

Progress Labels (1)

Progress labels are used to mark statements that signify effective progress. Any label that starts with progress is a progress label.

SPIN can be enabled to check for the existence of non-progress cycles, i.e., cycles that do not pass through at least one of the progress labels.

Important: If non-progress checking is enabled, then the search for invalid end states is automatically disabled.

Progress Labels (2)

```
byte x = 2;

active proctype A ()
{   do
    :: x = 3 - x; progress: skip
    od
}
active proctype B ()
{   do
    :: x = 3 - x
    od
}
```

UTA

Accept Labels

Accept labels are used to instruct SPIN to find all cycles that pass through at least one of the marked statements.

UTA

Never Claims (1)

... are used to specify behaviors that should never occur.

A never claim checks system properties just before and just after each statement execution.

If a never claim is matched by any system execution, then an error is triggered.

Never Claims (2)

A never claim can be used to check a system invariant p:

```
never {
  do
  :: ! p -> break
  :: else
  od
}
```

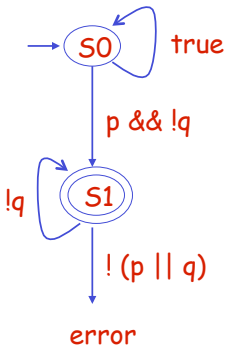

Never Claims (3)

Consider the following property: Every system state in which p is true eventually leads to a system state n which q is true, and in the interim p remains true.

In SPIN, we are interested in system executions that could violate the above property, i.e., executions in which p first becomes true and thereafter q either remains false forever, or p becomes false before q becomes true.

Never Claim (4)

```
never {
S0:  do
     :: p && !q -> break
     :: true
     od;
S1:
accept:
     do
     :: !q
     :: !(p || q) -> break
     od;
}
```



Never Claims (5)

What if we change the second alternative true to else in the first repetition?

Important: If we change true to else, then the never claim will only check the first time in a system execution that p becomes true should lead to a state where q is also true.

Never Claims (6)

As you may have felt, it is difficult to write never claims. SPIN provides a tool that can be used to convert a LTL formula to a never claim.

```
! [] (p -> (p U q))
```


Predefined variables (1)

There are four predefined variables in PROMELA:

- _ : a write-only variable
- _pid : the id of the current process
- np_ : indicates if a system is current in a progress state
- _last : the id of the process that performed the last step

UTA

Predefined Variables (2)

```
never { /* non-progress cycle detector */
    do
    :: true
    :: np_ -> break
    od;
accept:
    do
    :: np_
    od
}
```


UTA

Predefined Functions (1)

The following predefined functions are supposed to be used in never claims:

- pc_value (pid) – the current control state of process pid
- enabled (pid) – true if process pid has at least one transition enabled
- procname[pid]@label – returns a non-zero value if the next statement to be executed in process pid is marked with label.

UTA

Predefined Functions (2)

```
active proctype A ()
{
    printf ("%d\n", pc_value(_pid));
    printf ("%d\n", pc_value(_pid));
    printf ("%d\n", pc_value(_pid));
}
active proctype B ()
{
    pc_value (0) > 2 -> printf ("ok\n")
}
```


UTA

Putting together

- An assertion is used to specify a safety property that must hold at a particular point.
- An end label specifies alternative valid end states.
- A progress label marks a statement that signifies progress.
- An accept label is used to mark an acceptance state in a never claim.
- A never claim specifies properties that must be checked at each step.
- A trace assertion specifies valid or invalid sequences of channel operations.

Overview of SPIN

- Introduction
- SPIN by Examples
- The PROMELA Language
- Defining Correctness Claims
- Summary

Summary

- SPIN is designed to verify concurrent software systems.
- A PROMELA model consists of four types of basic objects. What are they?
- If a statement is blocked, the statement will be evaluated repeatedly until it becomes true.
- Correctness properties must be formally specified as part of a verification model.