CIS 721 - Real-Time Systems

Lecture 24: Verification Using SPIN

Mitch Neilsen
neilsen@ksu.edu

In Memory: Dr. Marvin Stone

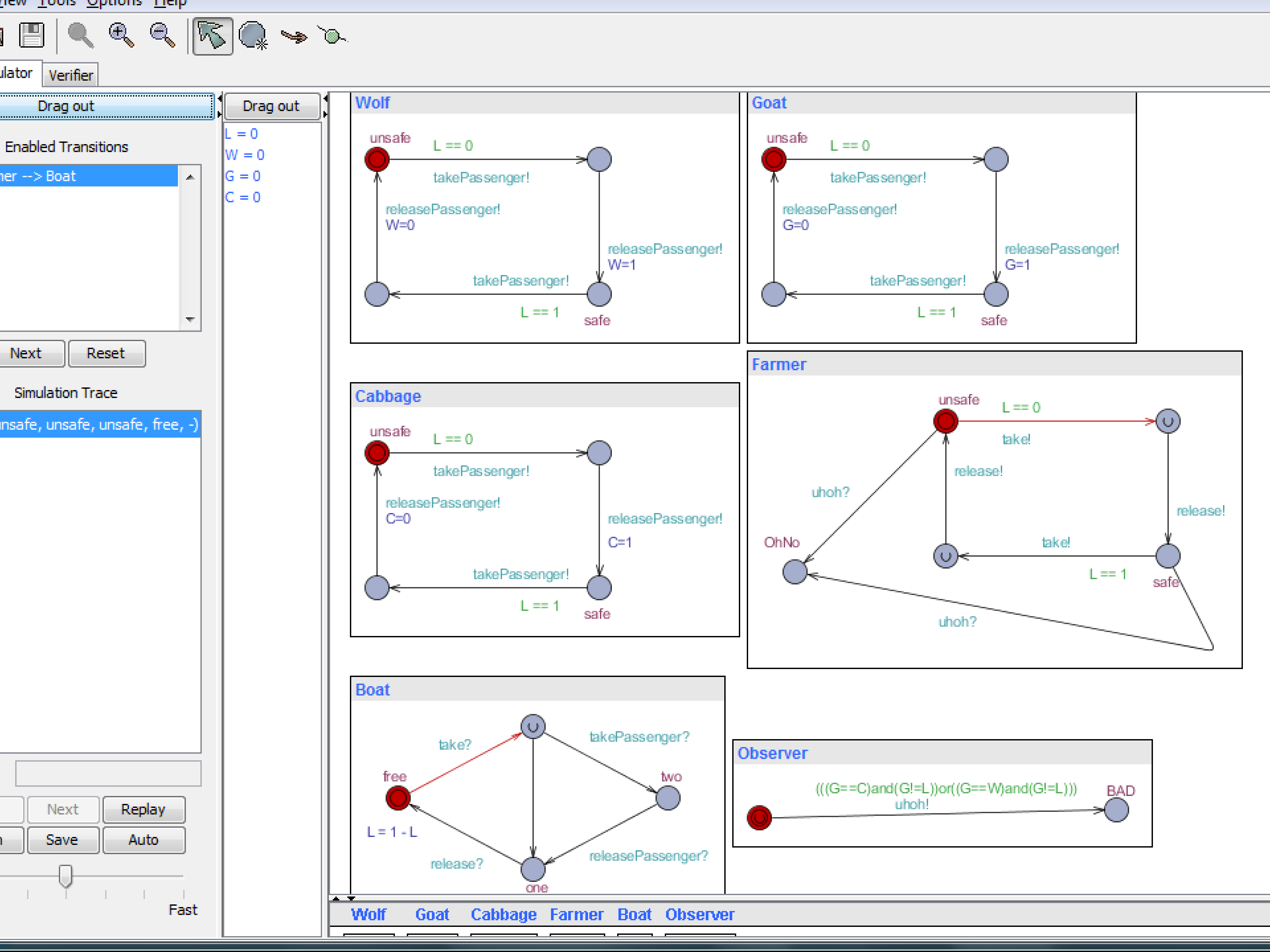
Student: Will you be teaching today?
Dr. Stone: No, I'll be lecturing. It's up to you to decide if I've taught you anything

Outline

- Verification and Validation Tools
  - **UPPAAL – Toolbox for validation and verification of real-time systems**
  - **Promela and SPIN**
    - **Simulation**
    - **Verification**
  - **Real-Time Extensions:**
    - **RT-SPIN – Real-Time extensions to SPIN**

UPPAAL Example

- Wolf, goat, cabbage, farmer problem
  - The farmer needs to move the wolf, goat, and cabbage from one side of the river to the other side. The farmer can only carry one passenger.
  - If the wolf and goat are left alone, the wolf will eat the goat. If the goat and cabbage are left alone, the goat will eat the cabbage.
  - How can the farmer transport the passengers without allowing one to be eaten?

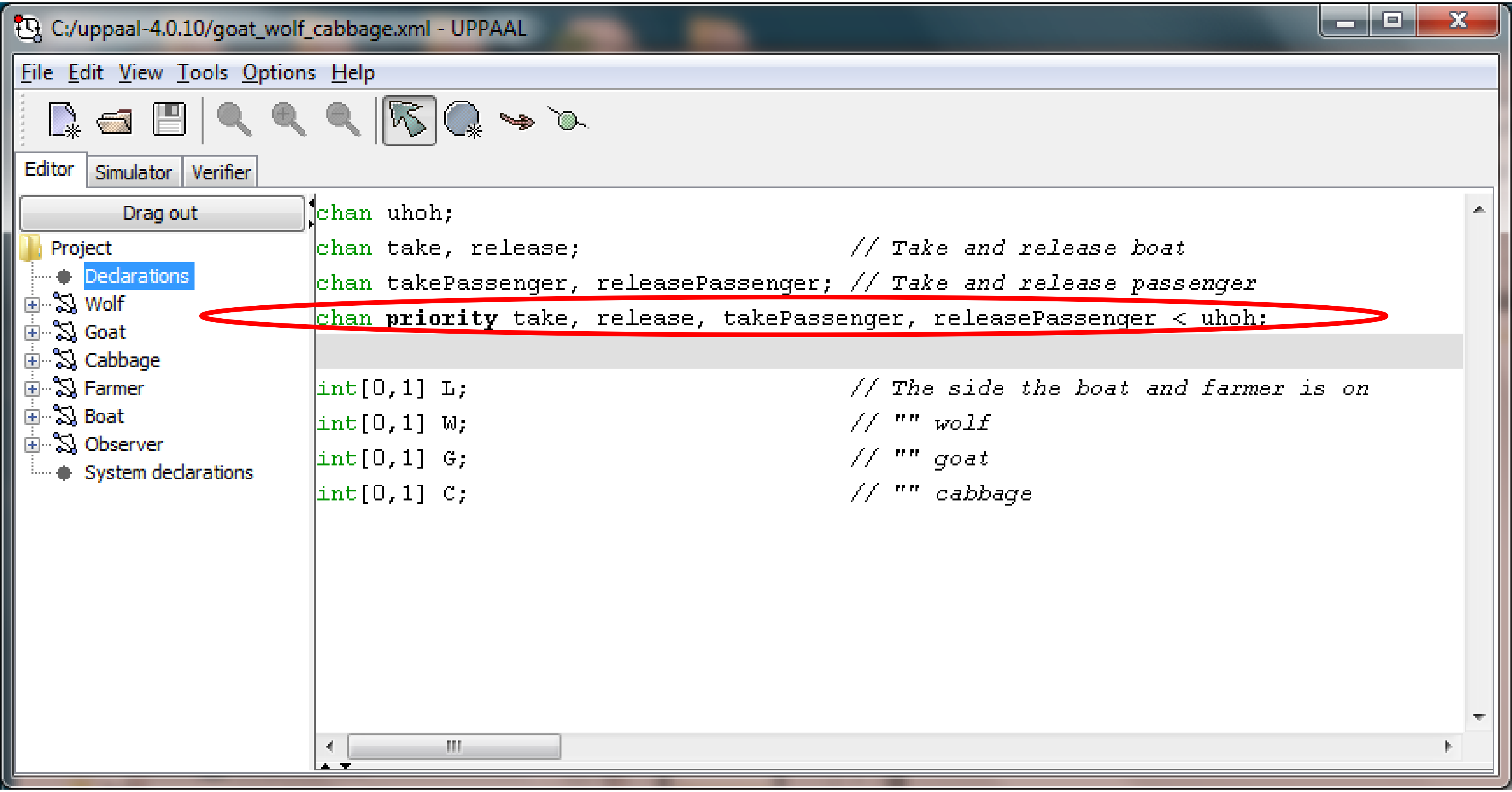


ner --> Boat

nsafe, unsafe, unsafe, free, -)

L = 0
W = 0
G = 0
C = 0

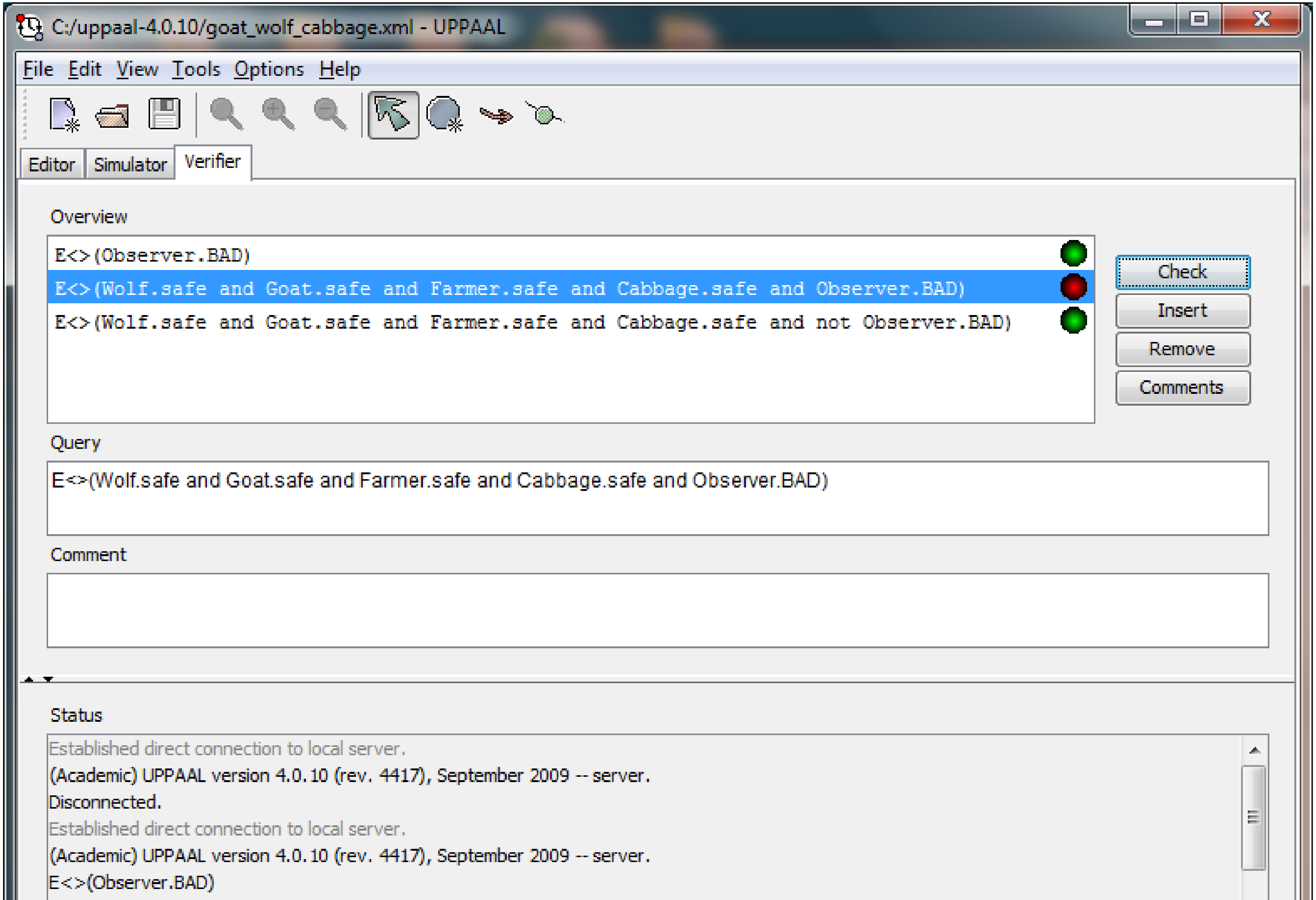
UPPAAL Model



```
chan uhoh;
chan take, release;                      // Take and release boat
chan takePassenger, releasePassenger; // Take and release passenger
chan priority take, release, takePassenger, releasePassenger < uhoh;

int[0,1] L;                              // The side the boat and farmer is on
int[0,1] W;                              // "" wolf
int[0,1] G;                              // "" goat
int[0,1] C;                              // "" cabbage
```

UPPAAL Model



```
E<>(Observer.BAD)
E<>(Wolf.safe and Goat.safe and Farmer.safe and Cabbage.safe and Observer.BAD)
E<>(Wolf.safe and Goat.safe and Farmer.safe and Cabbage.safe and not Observer.BAD)
```

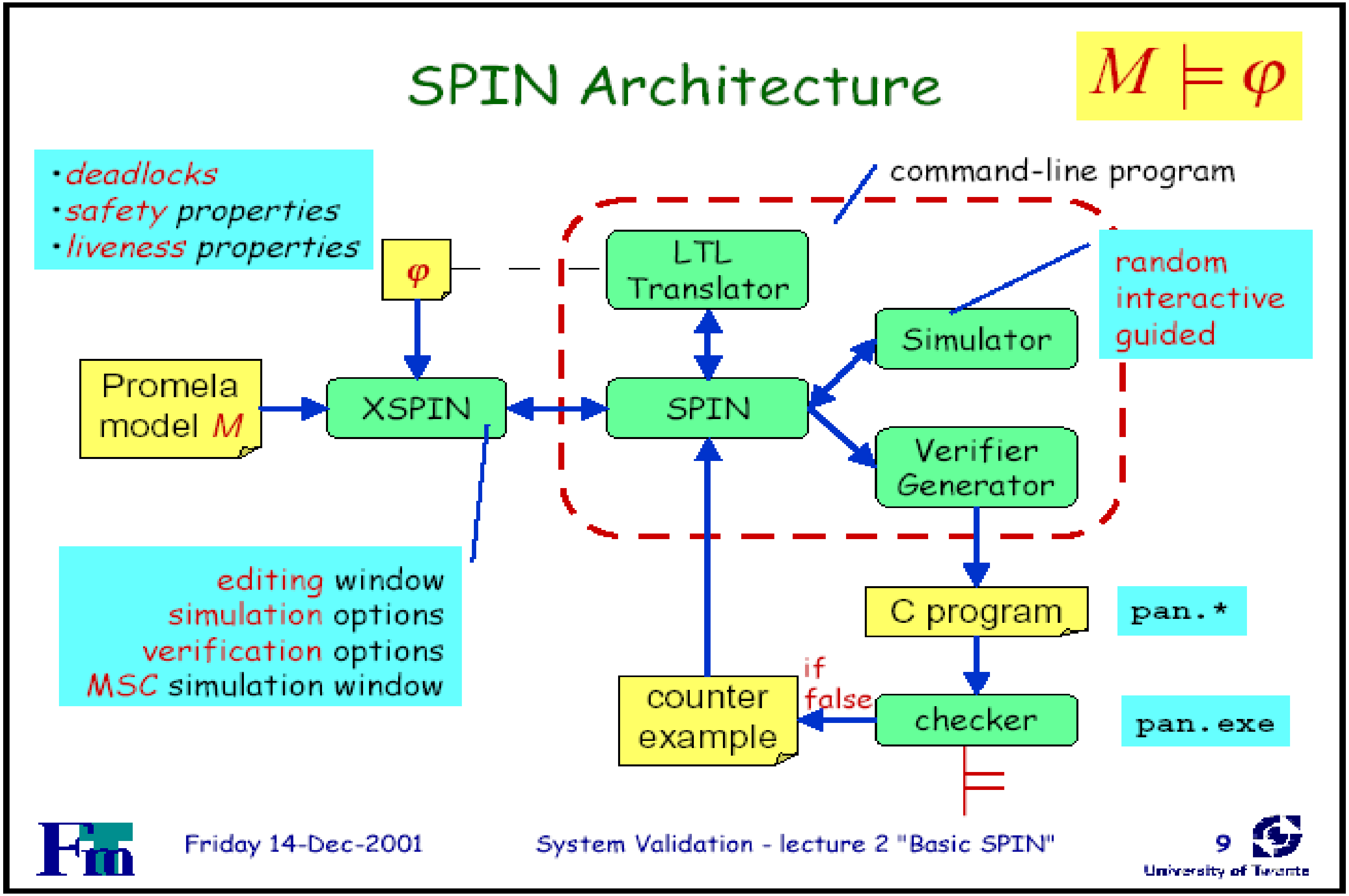
Query: E<>(Wolf.safe and Goat.safe and Farmer.safe and Cabbage.safe and Observer.BAD)

```
Established direct connection to local server.
(Academic) UPPAAL version 4.0.10 (rev. 4417), September 2009 -- server.
Disconnected.
Established direct connection to local server.
(Academic) UPPAAL version 4.0.10 (rev. 4417), September 2009 -- server.
E<>(Observer.BAD)
```

Model Checking

- **Problem:** The number of states can be very large.
- **Two Phases:**
  1. **Create a model:** some approximation of the system under construction; e.g., use a finite state model you need to model as well as its environment.
  2. **Verify the model:** determine the properties you want to verify, and check whether the model satisfies the properties.
- The verification exercise is only as good as the model.

Simple Promela Interpreter (SPIN)



SPIN Architecture

$M \models \varphi$

- *deadlocks*
- *safety* properties
- *liveness* properties

command-line program

random
interactive
guided

editing window
simulation options
verification options
MSC simulation window

Friday 14-Dec-2001

System Validation - lecture 2 "Basic SPIN"

University of Twente

Properties to Check using SPIN

- Deadlock
- Livelock, starvation
- Underspecification – Unexpected reception of messages
- Overspecification – Dead code
- Violations of constraints
  - Buffer overruns
  - Array bounds violations
- No assumptions are made about speed; e.g., testing **logical correctness** versus **real-time behavior**

Promela

- Promela – **Pro**cess/Protocol **Me**ta **La**nguage
  - Provides a language similar to the C programming language
  - Provides a guarded command language to model finite-state systems
  - Supports dynamic creation of concurrent processes
  - Supports messages channels between processes

Promela Model

- A Promela Model consists of the following elements:
  - type declarations
  - channel declarations
  - global variable declarations
  - process (proctype) declarations
  - [init process] - the init process is optional

Promela Statements

- skip - always executable
- assert(expression) - always executable
- assignment statements - always executable
- if statement - executable if at least one guard is
- do statement - executable if at least one guard is
- break statement - always executable
- send (ch!) - executable if channel ch is not full
- receive (ch?) - executable if channel ch is not empty

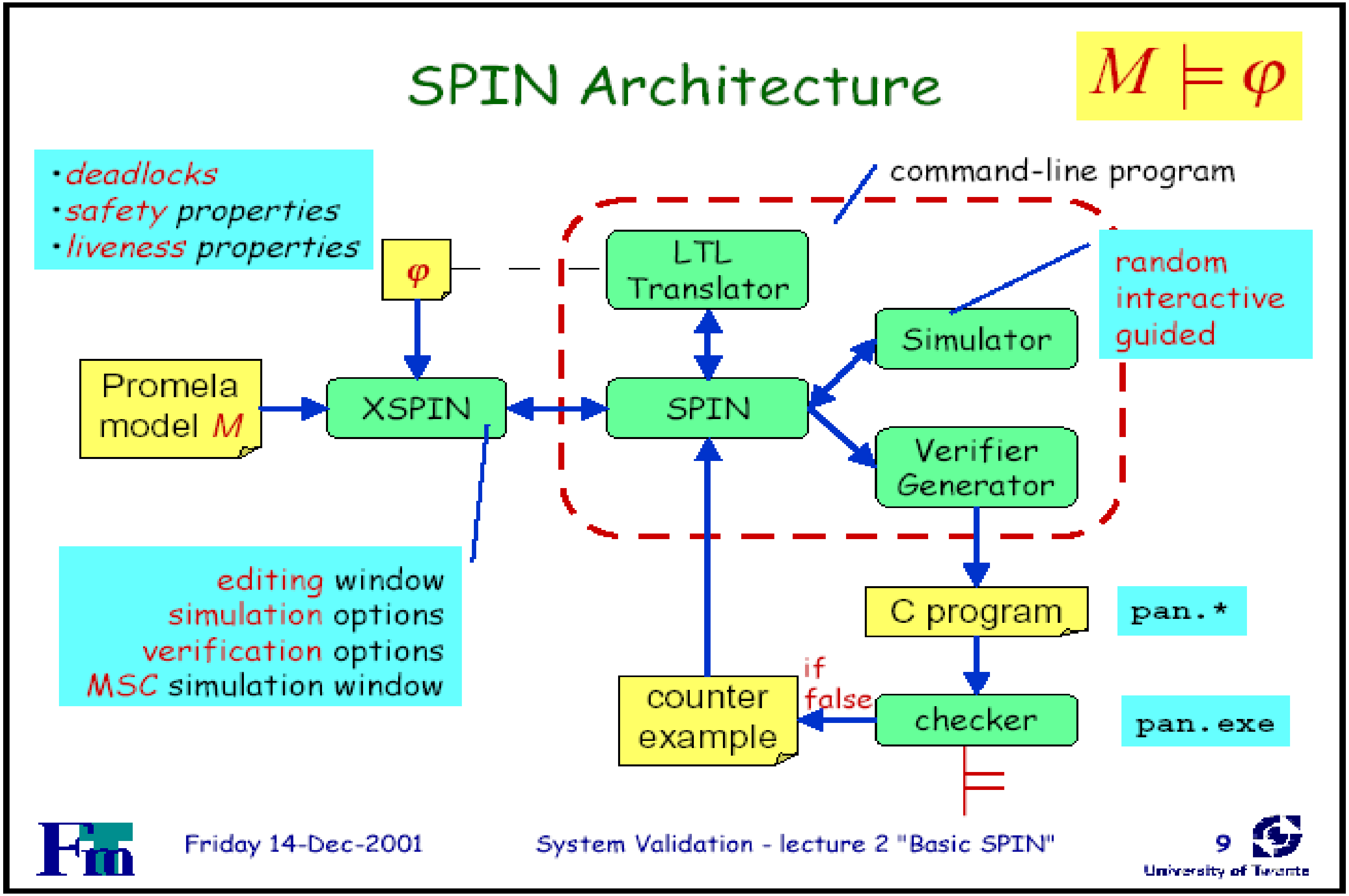
SPIN

- **SPIN** – **S**imple **P**romela **In**terpreter
  - A state-of-the-art model checking tool used to check the logical consistency of concurrent systems described using the modelling language Promela.
  - Designed specifically for checking data communication protocols.
  - http://spinroot.com/
  - http://spinroot.com/spin/Man/Exercises.html

XSPIN

- **XSPIN** is a high-level user interface that can be used to:
  - edit and check syntax of Promela models
  - simulate Promela models
  - verify Promela models
  - draw automata for each process in the model

SPIN Architecture



SPIN Architecture

$M \models \varphi$

- *deadlocks*
- *safety properties*
- *liveness properties*

Friday 14-Dec-2001 System Validation - lecture 2 "Basic SPIN" 9

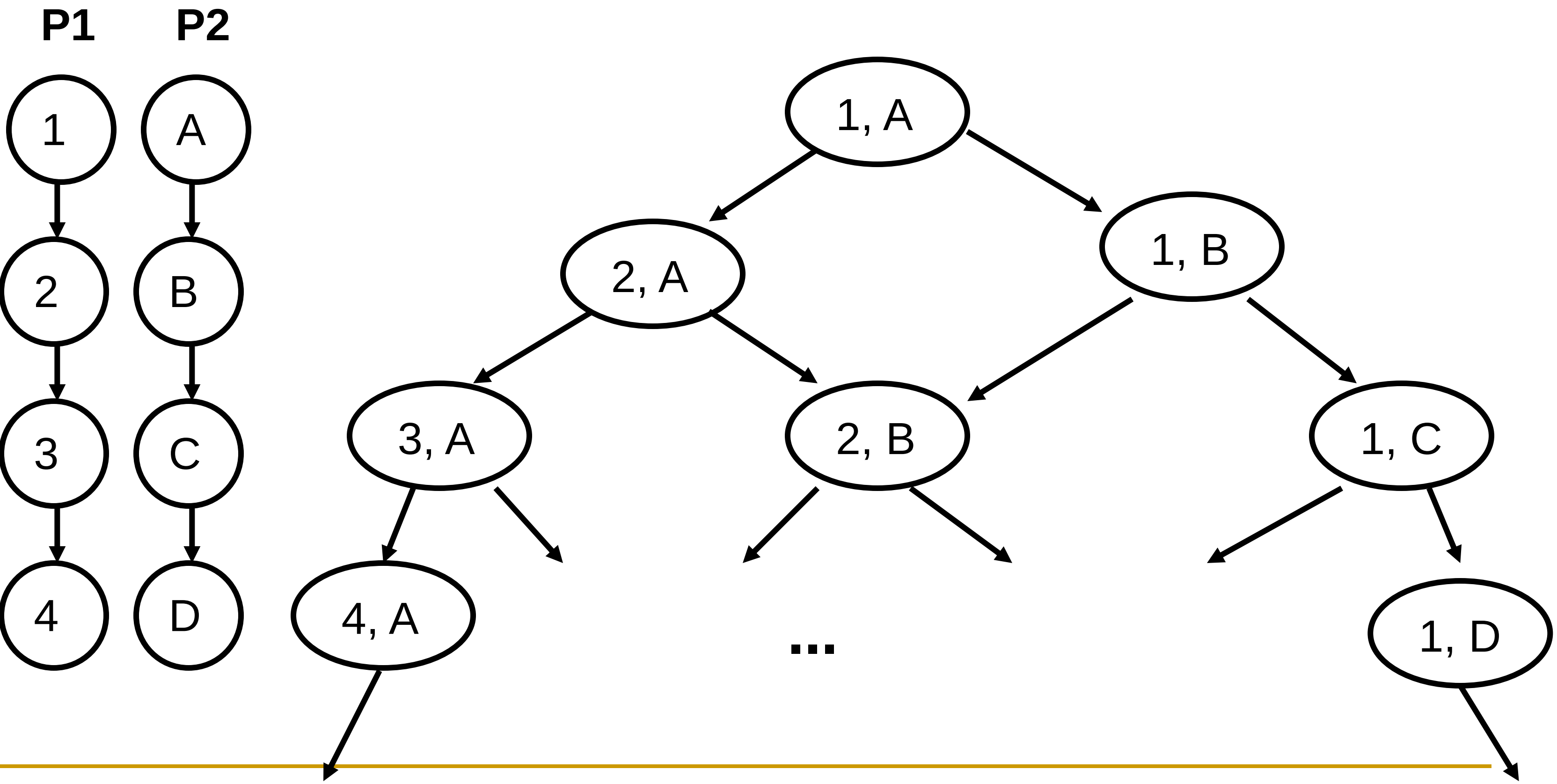
University of Twente

Concurrency

- SPIN processes execute **concurrently**.
- Processes are scheduled **non-deterministically**.
- Processes are interleaved, but statements are executed atomically.
- Each process may have several different possible actions (statements) enabled at each point in time, but only one action is (non-deterministically) selected to execute.

Example of Concurrency

- Processes P1 and P2 execute concurrently, k processes with n independent states each results in n^k global states.



if Statement

- If there is at least one guard (statement) that is executable, then the if statement is executable and SPIN non-deterministically selects one of the executable statements.
- If no guard is executable, then the if statement is blocked (not executable).
- The **->** operator is equivalent to ; . By convention, it is used to separate guards from statements.
- Example:

```
if
  :: guard one -> statement a; statement b; statement c;
  :: guard two -> statement d; statement e; statement f;
fi
```

Example: Random Number Generator

```
if
   :: skip -> n=1;
   :: skip -> n=2;
   :: skip -> n=3;
fi
```

do Statement

- With respect to choices, a do statement behaves just like an if statement.
- A do statement simply repeats the choice selection.
- The (always executable) break statement can be used to exit a do-loop.
- Example:

```
do
        :: guard one –> statement a;
        :: guard two –> statement b; break;
od
```

Example: Traffic Light

```
mtype = { RED, YELLOW, GREEN };
active proctype TrafficLight( )
{
   byte state = GREEN;
   do
       :: (state == GREEN) -> state = YELLOW;
       :: (state == YELLOW) -> state = RED;
       :: (state == RED) -> state = GREEN;
   od;
}
```

Channels

- Communication between processes is via channels, either for message passing or rendezvous (just set <dim> = 0 for a handshake).
- chan <name> = [<dim>] of { <type$_1$>, .., <type$_n$> }
  - <name> = name of the channel
  - <type$_i$> = type of elements to be transmitted
  - <dim> = maximum number of elements in the channel
- Example:
  - mtype = { DATA, ACK } ;
  - chan c = [5] of {mtype, bit};
  - sender executes: c ! DATA, 1;
  - receiver executes: c ? x, y; followed by: c ! ACK, y;

Example: Alternating Bit Protocol

- To every message, the sender adds a bit.
- The receiver acknowledges each message by sending the received bit back.
- The receiver only accepts messages with a bit that it expected to receive.
- If the sender is sure that the receiver has correctly received the previous message, it sends a new message and it alternates the accompanying bit.

Example: Alternating Bit Protocol

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

Example: Alternating Bit Protocol

```
proctype Receiver(chan in, out)
{
    bit recvbit;
    do
        :: in ? MSG, recvbit ->
                out ! ACK, recvbit;
        :: timeout ->
                out ! ACK, recvbit;
    od
}
init
{
    run  Sender(toSender, toReceiver);
    run  Receiver(toReceiver, toSender);
}
```

Promela Summary

- **A Promela Model consists of the following elements:**
  - **type** declarations – message types, typedefs, and constants
  - **channel** declarations – message channels
  - **global variable** declarations
  - **process** declarations
  - [**init** process] – initializes variables and starts processes

Promela Types

- **basic types**
  - bit, bool, byte, short, and int
- **mtype**
  - to define symbolic constants (usually used for messages)
  - mtype = { MSG, ACK, NACK, ERR }
- **typedefs**: to define structured types
- **arrays**: to gather variables of the same type
- **channels**: to support communication between processes
  - ! to send, ? to receive

Promela Statements

- **skip** is always executable
- **assert(**<expr>**)** is always executable
- an expression is executable if not zero
- an assignment statement is always executable
- **if** is executable if at least one guard is executable
- **do** is executable if at least one guard is executable
- **break** is always executable (exits **do**-statement)
- send **(ch!)** is executable if channel **ch** is not full
- receive **(ch?)** is executable if channel **ch** is not empty

Assert Statement

- **assert(A):** check whether assertion **A** is true.
- An **invariant** is an assertion that must be true in all states.
- Idea: create a **monitor** to ensure that the assertion remains true:

```
proctype monitor( )
{
   assert(A)
}
```

Mutual Exclusion - Example 1
(Incorrect Solution)

```
bit x1 = 0; /* used to indicate that process 1 wants in cs */
bit x2 = 0; /* used to indicate that process 2 wants in cs */
int mutex = 0;  /* used to count number of processes in cs */

proctype P1()
{
    x2 == 0;
    x1 = 1;
    mutex++;
      /* in critical section (cs) */
    mutex--;
    x1 = 0;
}

proctype P2()
{
    x1 == 0;
    x2 = 1;
    mutex++;
      /* in critical section (cs) */
    mutex--;
    x2 = 0;
}
```

```
proctype monitor()
{
    assert(mutex!=2);
}

init
{
    run P1();
    run P2();
    run monitor();
}
```

Mutual Exclusion - Example 2

(Peterson's Solution)

```
bit x1 = 0; /* used to indicate that process 1 wants in cs */
bit x2 = 0; /* used to indicate that process 2 wants in cs */
int mutex = 0;  /* used to count number of processes in cs */
int turn = 0;   /* indicates whose turn it is to enter cs  */

proctype P1()
{
    x1 = 1;
    turn = 2;
    (x2 == 0) || (turn == 1);
    mutex++;
      /* in critical section (cs) */
    mutex--;
    x1 = 0;
}
proctype P2()
{
    x2 = 1;
    turn = 1;
    (x1 == 0) || (turn == 2);
    mutex++;
      /* in critical section (cs) */
    mutex--;
    x2 = 0;
}
```

```
proctype monitor()
{
    assert(mutex!=2);
}

init
{
    run P1();
    run P2();
    run monitor();
}
```

Mutual Exclusion

(Peterson's Solution Revisited)

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

active [2] proctype user()
{
     assert(_pid == 0 || _pid == 1);
again:
     flag[_pid] = 1;
     turn = _pid;
     (flag[1 - _pid] == 0 || turn == 1 - _pid);
     ncrit++;
         assert(ncrit == 1);   /* critical section */
     ncrit--;
     flag[_pid] = 0;
     goto again
}
```

Macros – Preprocessor (cpp)

- Promela uses **cpp**, the C preprocessor to preprocess Promela models which is useful to define:
  - **Constants**: #define MAX 4
  - **Macros**: #define RESET_ARRAY(a) \
    d_step { a[0]=0; a[1]=0; a[2]=0; a[3]=0; }
  - **Conditional model fragments**:

```
#define LOSSY 1
#ifdef LOSSY
    active proctype Deamon() {/* steal messages */}
#endif
```

- All **cpp preprocessor** commands start with a hash:
  #define, #ifdef, #include, etc.

Inline procedures

- Promela also has its own **macro-expansion** feature using the **inline** construct.

```
inline init_array(a) {
    d_step {
        i=0;
        do
          :: i<N -> a[i] = 0; i++;
          :: else -> break;
        od;
        i=0;
    }
}
```

- **Notes:** all variables should be declared elsewhere; also, reset local counter variables (e.g., i = 0), inline procedures cannot be used as an expression.

Unless statement

- **{ <statements> } unless { guard; <statements> }**
- Statements in <statements> are executed until the first statement (guard) in the escape sequence becomes executable, then the other statemenst are executed. This resembles exception handling in languages like Java.
- **Example:**

```
proctype Processor()
{
  {
    /* execute these statements */
    …
  } unless { port ? INTERRUPT; /* other statements */ }
}
```

Timeout

- Promela does **not** have real-time features.
  - In Promela we can only specify functional behavior.
  - Most protocols, however, use timers or a timeout mechanism to resend messages or acknowledgements.
- **Timeout**
  - In SPIN, **timeout** becomes executable if there is no other process in the system which is executable.
  - **timeout** models a global timeout and provides an escape from deadlock states.
  - Note: Beware of statements that are always executable.

Safety Properties

- A **safety property** is used to check if "nothing bad ever happens".
- **Examples:**
  - **invariants**: x is always less than some constant
  - **deadlock freedom**: the system never reaches a state where no actions are executable
  - **mutual exclusion**: the system never reaches a state where two processes are in the critical section.
- SPIN tries to find a trace leading to the "bad" thing.
- If no such trace exists, then the property is satisfied.

Liveness Properties

- A **liveness property** is used to check if "something good will eventually happen".
- **Examples:**
  - termination: "the system will eventually terminate"
  - response: "if action X occurs, then action Y will occur eventually"
- SPIN tries to find a (infinite) loop in which the "good" thing does not happen. If there is no such loop, then the property is satisfied.

Linear Temporal Logic (LTL)

- LTL formulae are used to specify liveness properties.
- LTL includes propositional logic and temporal operators:
  - **[ ]P =** always P
  - **<>P =** eventually P
  - **P U Q =** P is true until Q becomes true
- **Examples:**
  - Invariance: **[ ] (p)**
  - Response: **[ ] ((p) -> (<> (q)))**
  - Precedence: **[ ] ((p) -> ((q) U (r)))**
  - Objective: **[ ] ((p) -> <>((q) || (r)))**

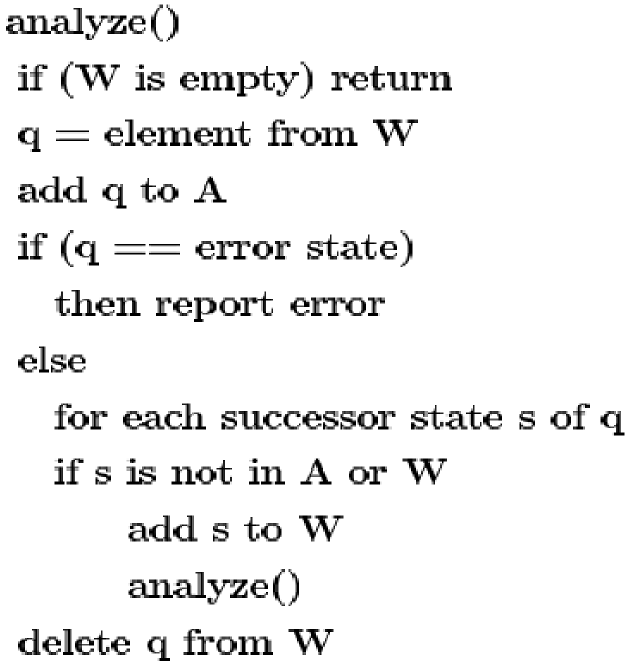
Properties

Properties that can be checked with SPIN:

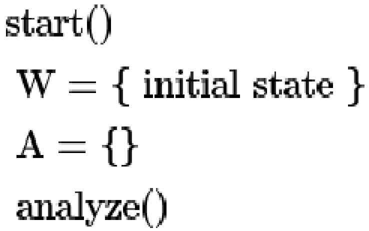
- deadlocks (invalid endstates)
- assertions
- unreachable (dead) code
- LTL formulas
- liveness properties
  - non-progress cycles (livelocks)
  - acceptance cycles

Simulation Algorithm

- Execution sequence: s_0, s_1, s_2, …
  - where s_0 is the initial state and
  - s_i follows from s_{i-1} in the execution sequence.
- **Algorithm:**



```
analyze()
 if (W is empty) return
 q = element from W
 add q to A
 if (q == error state)
    then report error
 else
    for each successor state s of q
    if s is not in A or W
          add s to W
          analyze()
 delete q from W
```



```
start()
 W = { initial state }
 A = {}
 analyze()
```

Lossy Channel

- To model a lossy channel, simply add a message stealing daemon.
- How can we be sure that the protocol works correctly when messages are lost?
  - Model different messages with a sequence number.
  - Assert that the protocol works correctly.

SPIN Algorithm

- SPIN uses a depth first search algorithm (DFS) to generate the complete state space (Statespace).

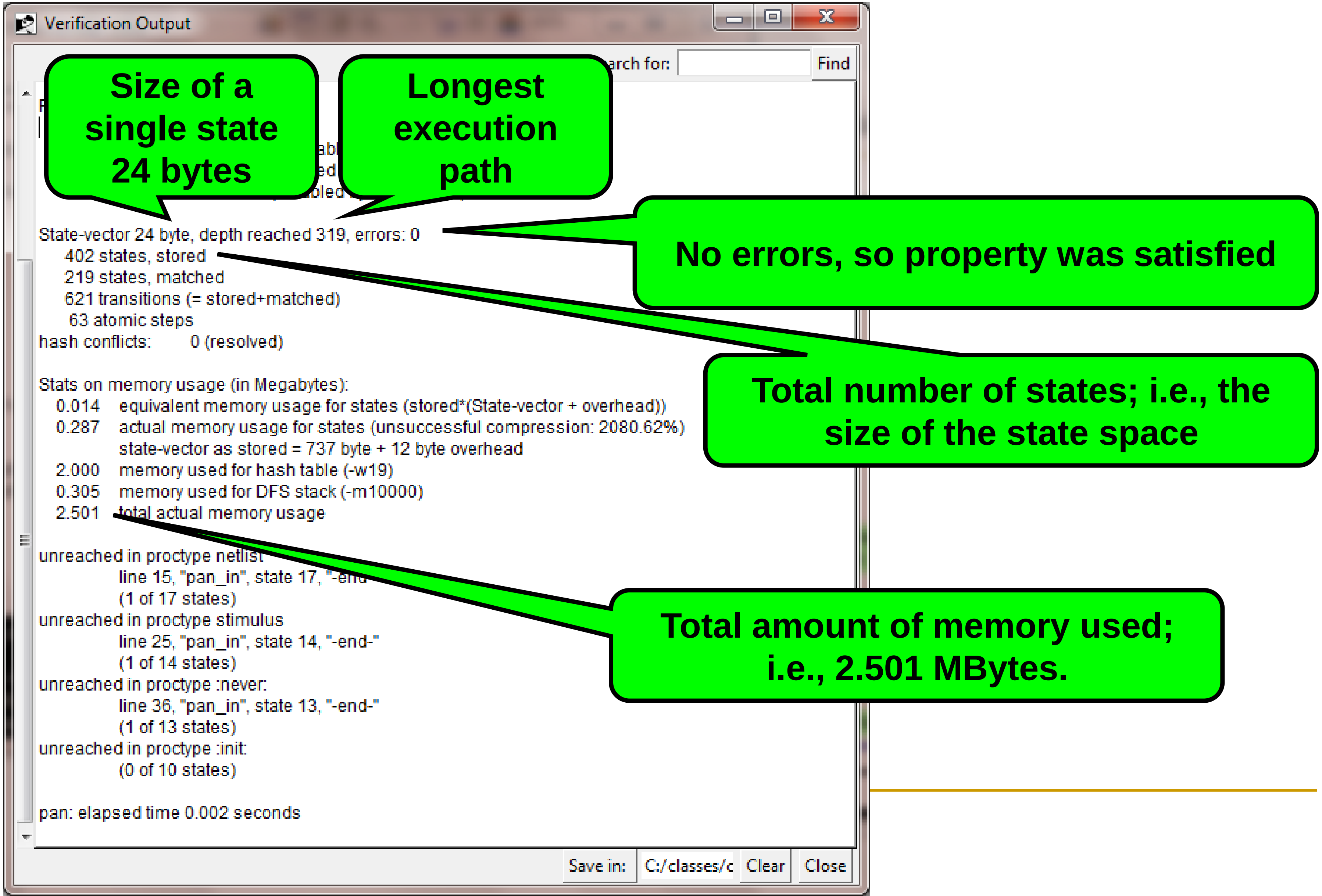
```
procedure dfs(s: state) {
  add s to Statespace;
  if error(s)  reportError();
  foreach (successor t of s) {
      if (t not in Statespace)
      dfs(t)
  }
}
```

- Note that tree construction and error checking is performed at the same time; SPIN is an **on-the-fly model checker**.
- States are stored in a hash table, and old states are stored on a stack.

State Vector

- A state vector is the information to uniquely identify a system state; it contains:
  - global variables
  - contents of the channels
  - for each process in the system:
    - local variables
    - process counter of the process
- For efficient modelling, it is important to minimize the size of the state vector.

SPIN Verification Report



```
State-vector 24 byte, depth reached 319, errors: 0
    402 states, stored
    219 states, matched
    621 transitions (= stored+matched)
     63 atomic steps
hash conflicts:        0 (resolved)

Stats on memory usage (in Megabytes):
   0.014    equivalent memory usage for states (stored*(State-vector + overhead))
   0.287    actual memory usage for states (unsuccessful compression: 2080.62%)
            state-vector as stored = 737 byte + 12 byte overhead
   2.000    memory used for hash table (-w19)
   0.305    memory used for DFS stack (-m10000)
   2.501    total actual memory usage

unreached in proctype netlist
            line 15, "pan_in", state 17, "-end-"
            (1 of 17 states)
unreached in proctype stimulus
            line 25, "pan_in", state 14, "-end-"
            (1 of 14 states)
unreached in proctype :never:
            line 36, "pan_in", state 13, "-end-"
            (1 of 13 states)
unreached in proctype :init:
            (0 of 10 states)

pan: elapsed time 0.002 seconds
```

Size of a single state 24 bytes

Longest execution path

No errors, so property was satisfied

Total number of states; i.e., the size of the state space

Total amount of memory used; i.e., 2.501 MBytes.

Typical Checks

Several checks are typically used to test for properties: deadlock, assertions, invariance, and liveness (LTL):

1. **Sanity check** – random and interactive simulations
2. **Partial check** – use SPIN's bitstate hashing (states are not stored) mode to quickly sweep over the state space.
3. **Exhaustive check** – if bitstate hashing fails, SPIN supports several options to proceed:
    - Compression of state vector
    - Optimization (SPIN options or manual)
    - Abstractions (manual)
    - Bitstate hashing

SPIN Optimization Algorithms

- Several optimization algorithms are available to make SPIN runs more efficient:
    - partial order reduction
    - minimized automaton encoding of states
    - state vector compression
    - bitstate hashing
- SPIN supports many command-line options to activate and tune these optimization algorithms
- For example: Xspin -> Run -> Set verification params -> Set advanced options -> Extra compile-time directives

Modelling Considerations

Space vs. time considerations:

- Number of states
- Size of the state vector
- Maximum search depth
- Verification time

Multiple validation models:

- Worst case: one model for each property.
- This is different than general programming where developers only design a single program.

Example: Pure Atomicity

- To test that none of the atomic clauses in the model are ever blocked; e.g., pure atomicity:
  - Add a global variable: bit flag;
  - Change all atomic clauses to:

```
atomic {
  stmt1;
  flag = 1;
  ..
  stmtn;
  flag = 0;
}
```

  - Check that flag is always 0: **[ ] !flag**

Invariance

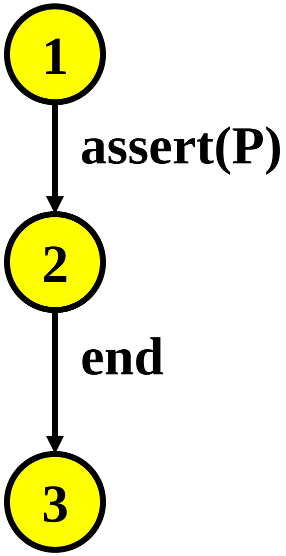
- **Always P = []P** where **P** is a state property:
  - safety property
  - invariance = global universality or global absence
- Approximately 25% of the properties typically being checked with model checkers are invariance properties, and 48% of the properties are response properties; e.g.:
  - **[] !flag**
  - **[] mutex < 2**
- SPIN supports several ways to check for invariance.

1,2. Monitor process (single assert)

- Proposed in Spin's documentation
- Add the following monitor process to the Promela model:

```
active proctype monitor()
{
   assert(P);
}
```

- Two **variations**:
  1. monitor process is created **first**
  2. monitor process is created **last**



3. Guarded monitor process

- Drawback of solution “1+2 monitor process” is that the **assert** statement is enabled in every state.

```
active proctype monitor( )
{
    assert(P) ;
}
```

⇒

```
active proctype monitor( )
{
    atomic {
        !P -> assert(P);
    }
}
```

- The **atomic** statement only becomes executable when P itself is **not** true.

4. Monitor process (do assert)

- From an operational perspective, the following monitor process seems less effective, but there are fewer states:

```
active proctype monitor( )
{
   do
      :: assert(P)
   od
}
```

Checking Invariance

- Experimentally, methods 1 and 2 perform the worst -- when checking invariance, these methods should be avoided.
- Method 4 “monitor do assert” performs well, but may change the model if it contains a timeout; e.g., the do-assert loop is always executable, so a timeout will never be executed.
- Overall, method 3 “guarded monitor process” is the most effective and reliable for checking invariance.

Rules of Thumb
(How to construct an efficient Promela model)

- **Data and variables:**
  - All data ends up in the state vector.
  - More states are generated if a variable can be assigned more values – limit variable size.
  - Limit channel size (e.g., the channel dimension).
  - Prefer local variables over global variables.
- **Atomicity:**
  - Enclose statements that do not need to be interleaved with atomic or d_step statements.
  - Beware of infinite loops or other semantic changes due to restrictions in interleaving.
- **Processes:**
  - If possible, combine the behavior of two processes into a single process.

Verification

- **Verification** means proving correctness; that is, establishing that a design fullfills certain properties of interest (assertions) or that a particular property will never be satisfied (a never claim).
- **Why is verification needed?**
  - The proliferation of embedded systems is widespread.
  - System reliability depends on correct functioning of both hardware and software.
  - Embedded systems are used in safety-critical control systems in which errors can be fatal or very costly.

Verification versus Testing

- Testing starts with a set of possible test cases, simulates the system on each input, and observes the behavior. In general, testing does not cover all possible executions.
- On the other hand, verification establishes correctness for **all** possible execution sequences.

Techniques for Verification

- **Formal verification**: prove mathematically that the program is correct – this can be difficult for large programs.
- **Correctness by construction**: follow a well-defined methodology for constructing programs.
- **Model checking**: enumerate all possible executions and states, and check each state for correctness.

Summary

- Next Time
  - **Hardware Model Checking using SPIN**
  - **RT-SPIN – Real-Time extensions to SPIN**