
Design and Validation of Network Protocol

A Need for Formal Methods

Dario Vieira
EFREI - 2014

Procedure Rules

Protocol Design

- An important aspect of the protocol design problem is that the procedure rules are interpreted concurrently by a number of interacting processes
 - The effect of each new rule we add to the set is often much larger than can be foreseen
 - Many different interleavings in time of the interpretation of these rules by the various processes will be possible
 - Precisely because of this concurrency a protocol behavior is not always reproducible
 - To convince ourselves of the correctness of a design we need something better than informal reasoning
-

Procedure Rules

Protocol Design

- We must be able to express behavior unambiguously in a convenient formal notation
 - Transition tables, Formal Finite State Machines, MSC (Message Sequence Chart) or SDL (Specification and Description Language) diagrams, for instance, can be used for this purpose
 - There is no general methodology that can guarantee a priori the design of an unambiguous set of procedure rules
 - There are, however, tools with which we can, even automatically, verify the logical consistency of the rules and the observance of the correctness requirements
 - And, of course, there is common sense and plain good engineering practice that can help us to keep the protocol rules manageable
-

Formal Languages

- Components of a formal language:
 - An alphabet;
 - Strings; and
 - Operations
- An **alphabet** is a finite collection of symbols
 - Symbols in alphabet cannot be decomposed
 - Example symbols: a, b, left, right, move_robot_arm, etc
- A **string** (or word) is a finite sequence of zero or more symbols taken from an alphabet
 - A string containing no symbol is called an empty string
- Some useful **operations** of strings
 - Examples: length of a string, join of two strings, index of a symbol in a string, etc.

Formal Languages (Cont'd)

- A formal language is any set of strings that can be formed using symbols of an alphabet
 - A language whose strings are formed from symbols in an alphabet **A** is said to be a language over **A**
- The strings in a language are called the **sentences** of that language
- The symbol **A*** is used to represent all possible strings over the alphabet **A**. So any language **L** over **A** is a subset of **A***

How Design Problem Can be Solved

- We need effective methods for:
 - Designing and describing protocols
 - Checking that a protocol submitted for standardization is correct
- In other words, we need to be able to:
 - Express a design criteria of a protocol
 - Examples (Holzmann 91):
 - Absence of any unwanted behaviors of a system, such as the infinite repetition of actions that do not meet a progress criterion defined by the user
 - Guaranteed observance of a specific functionality, e.g., within finite time after lifting the receiver the telephone subscriber must receive a dial-tone
 - Check that its design criteria are met
- Protocol specification languages have been developed
 - Referred to as: **Formal Description Techniques** (FDTs)
 - Used for **specifying protocols**

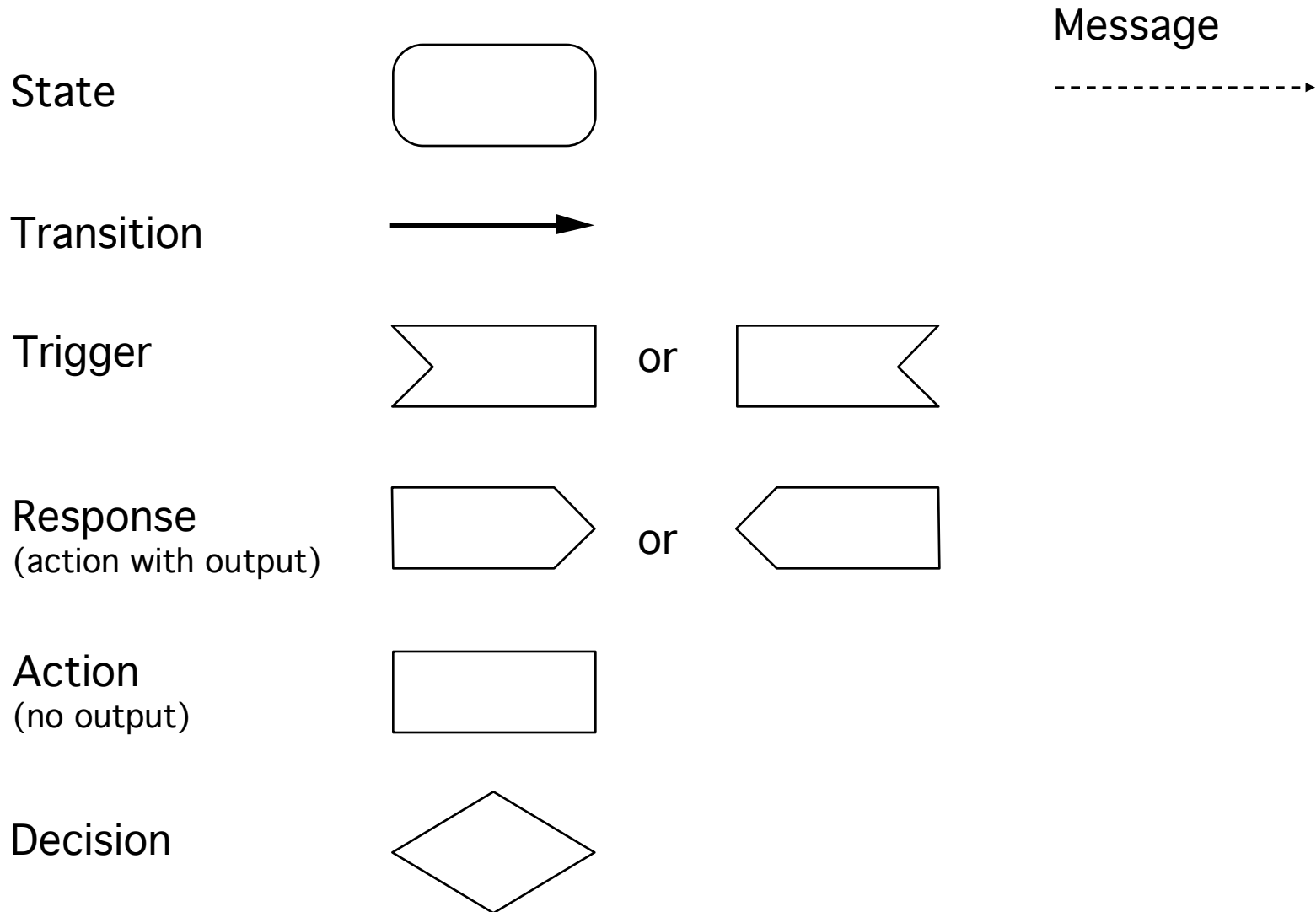
Formal Description Techniques

- Three FDTs have been standardized:
 - SDL
 - LOTOS
 - Estelle

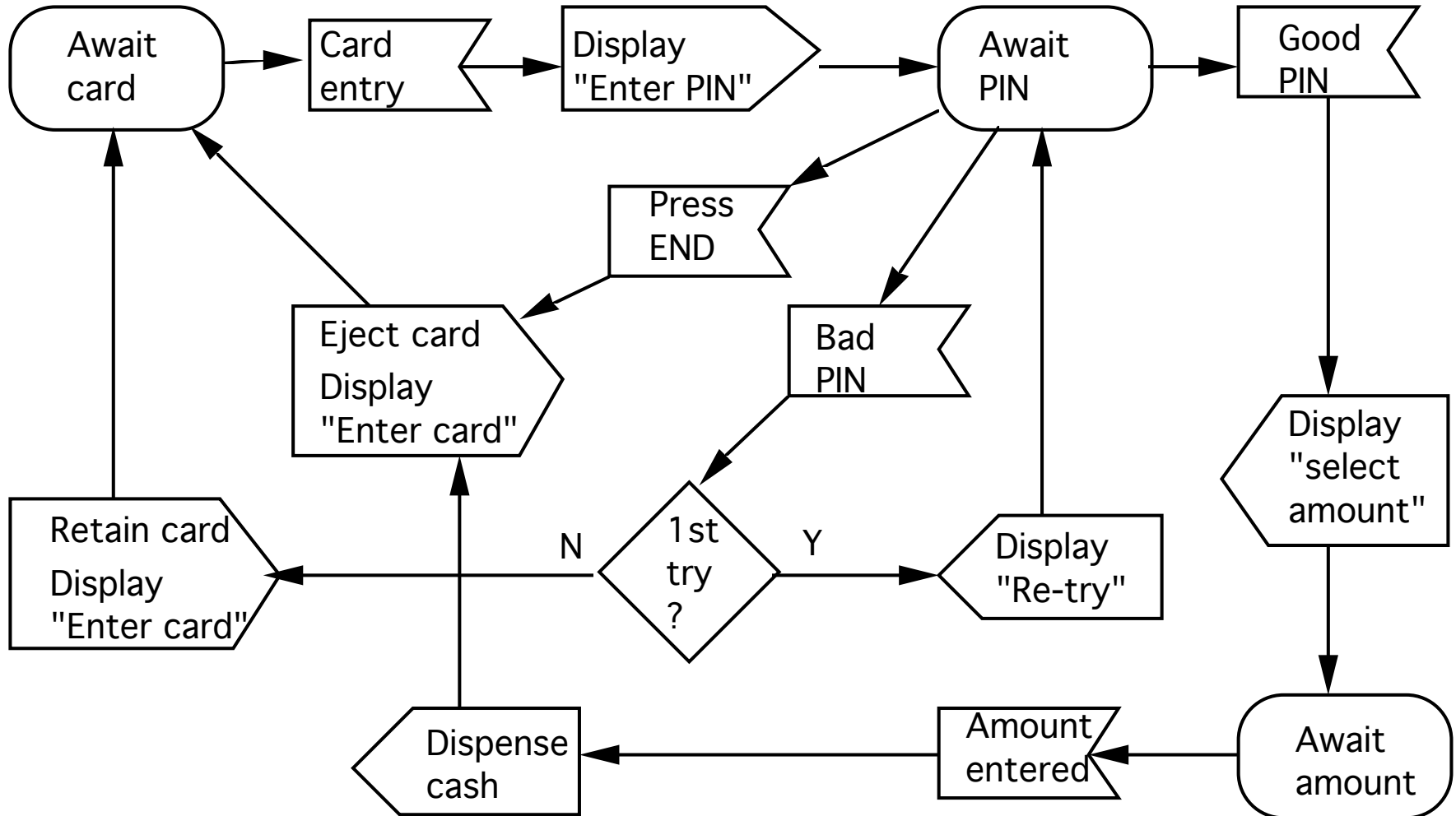
Specification and Description Language (SDL)

- SDL was developed by a CCITT study group
 - Work started in 1968
 - A final stable version was approved in 1987
- Meant specifically for the specification and design of telecommunication systems, e.g., telephone switches
- Two variants of SDL:
 - Graphical form
 - Program form

SDL



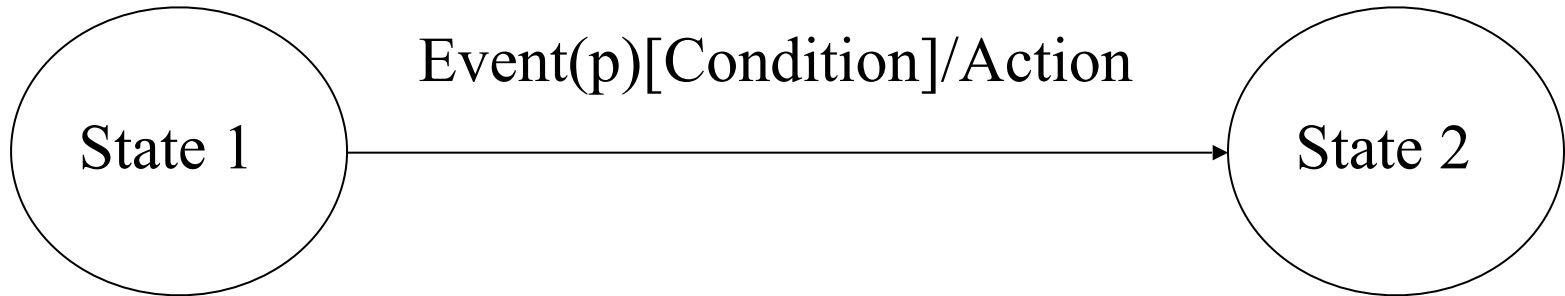
SDL - Guichet bancaire



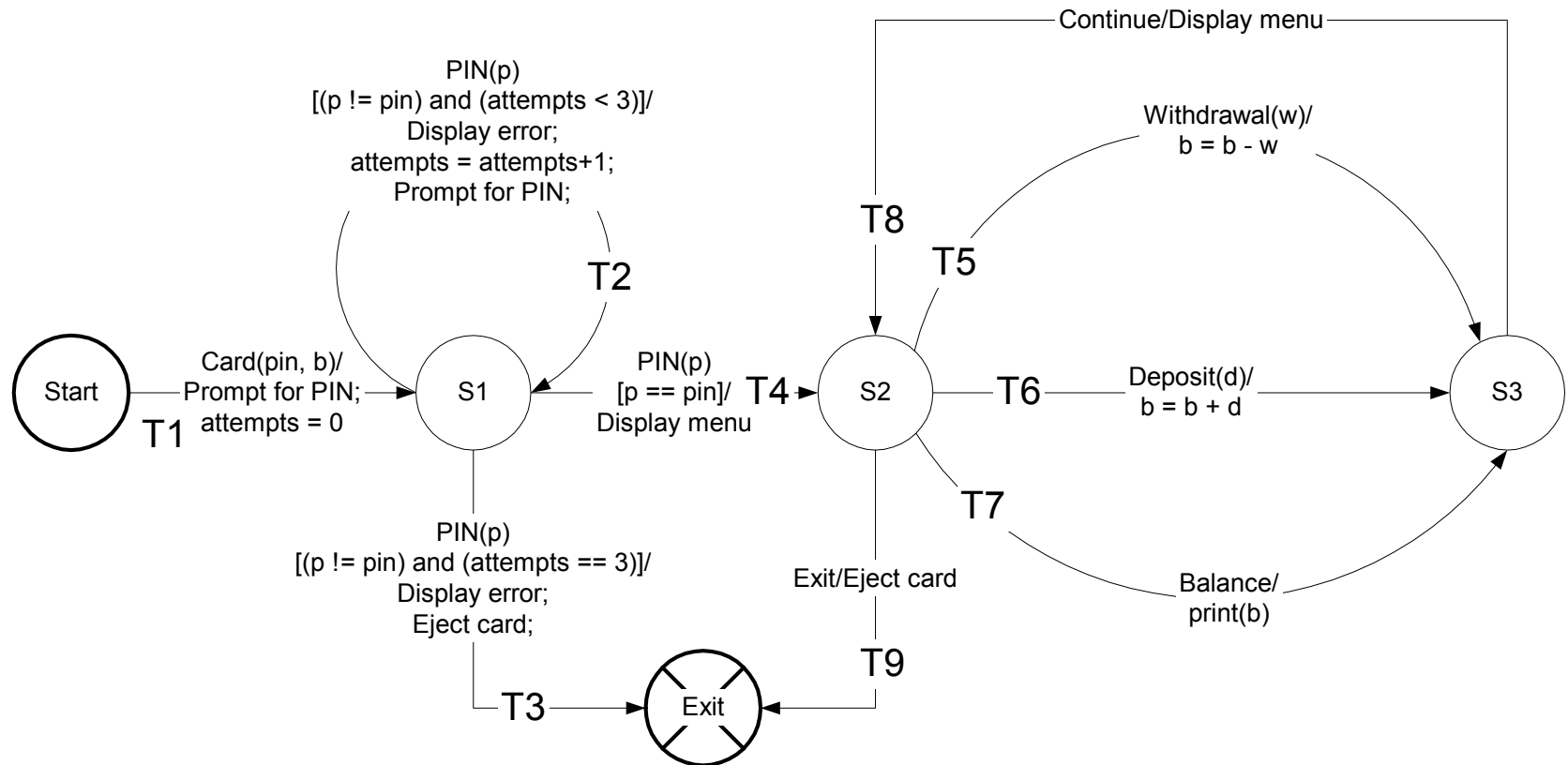
Estelle

- Being developed under another ISO subgroup
 - Issued as international ISO standard in 1989
- Based on an Extended Finite State Machine (EFSM) concept
- Extended Finite State Machine (EFSM)
 - FSM-Block: a conventional FSM realizing the state transition graph of the EFSM model
 - A-block: an arithmetic block for performing the data operation associated with each transition. The operation of this block is regulated by the output signal of the FSM block
 - E-block: a block for evaluating the trigger conditions associated with each transition. The input of this block are the data variables

EFSM Transition



EFSM Model of the ATM System



Language Of Temporal Ordering Specification (LOTOS)

- It is being developed within ISO
 - Issued as international standard in 1989
- LOTOS is also called a process algebra
 - Based on the **calculus of communicating systems** (CCS) developed by Robin Milner at University of Edinburgh
- Main goal of process algebra:
 - Formal specification of process behaviors on a high level of abstraction
 - Algebra defines:
 - a rigorous set of transformations rules, and
 - equivalence relations
 - Allows a designer to reason formally about behaviors

Basic LOTOS

– La structure du langage :

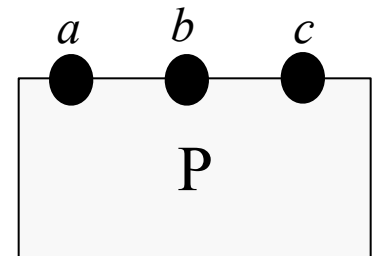
process *<process_id>* *<parameter_part>* :=
 <behaviour_expression>
endproc

Nom du processus

Liste des portes du processus

Expression régulière définissant le comportement du processus

Exemple : **process** P [a, b, c] := ... **endproc**



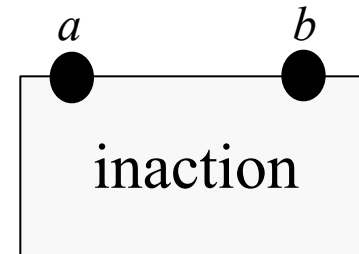
Basic LOTOS

Pour continuer...

Les opérateurs et constructions du langage

- Le processus élémentaire **stop**
Processus qui ne peut rien faire

```
process inaction [a, b] :=  
    stop  
endproc
```



=> **stop** est assimilé à un **deadlock**

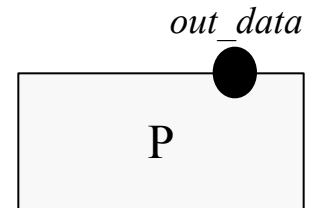
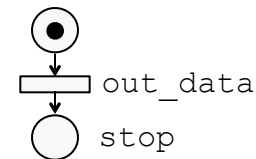
Basic LOTOS

Les opérateurs et constructions du langage

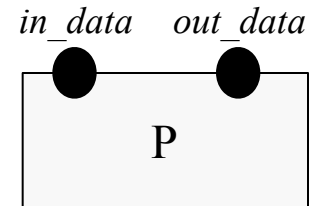
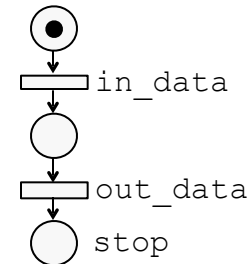
- **préfixage** : **a ; B**
 - a est une action atomique
 - B est une expression de comportement

Exemple :

```
process P [out_data] :=  
    out_data ; stop  
endproc
```



```
process Q [in_data, out_data] :=  
    in_data ; out_data ; stop  
endproc
```



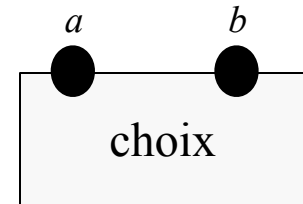
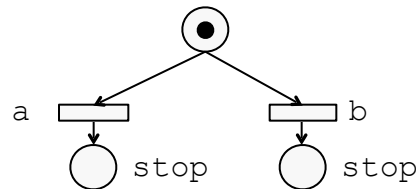
Basic LOTOS

Les opérateurs et constructions du langage

- **Choix non déterministe** : $B1 \ [] \ B2$
 - B1 et B2 sont des expressions de comportement
 - $B1 \ [] \ B2$ se comporte comme B1 ou comme B2 selon le comportement de l'environnement du processus ou de façon non-déterministe
 - si l'environnement offre l'action initiale de B1, alors B1 est sélectionné
 - si l'environnement offre l'action initiale de B2, alors B2 est sélectionné
 - Axiomes :
 - $B1 \ [] \ B2 = B2 \ [] \ B1$ (commutativité)
 - $B \ [] \ B = B$ (idempotence)
 - $B1 \ [] \ (B2 \ [] \ B3) = (B1 \ [] \ B2) \ [] \ B3$ (associativité)
 - ...

Exemple :

```
process choix [a, b] :=  
    a ; stop  
    []  
    b ; stop  
endproc
```



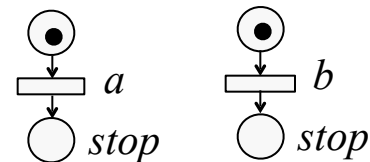
Basic LOTOS

Les opérateurs et constructions du langage

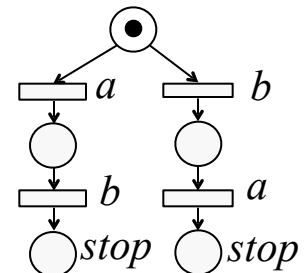
- **Parallélisme sans interaction** : $B1 \parallel B2$
 - $B1$ et $B2$ sont des expressions de comportement
 - $B1 \parallel B2$ se comporte comme $B1$ et comme $B2$ **sans interactions**
 - Axiomes :
 - $B1 \parallel B2 = B2 \parallel B1$
 - $B1 \parallel (B2 \parallel B3) = (B1 \parallel B2) \parallel B3$
 - $B \parallel \text{stop} = B$
 - $(a; B1) \parallel (b; B2) = a; (B1 \parallel (b; B2)) \sqcup b; ((a; B1) \parallel B2)$
 - ...

Exemple :

```
process parallel [a, b] :=  
    a ; stop  $\parallel$  b ; stop  
endproc
```



équivalent à

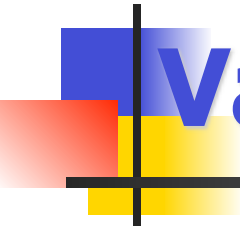


Limitations of FDTs

- LOTOS is the only language that also addresses the design issues, in addition to specification
- None of the FDTs addresses the problem that complete designs must be verifiable at the protocol specification level
 - We must be able to check with automated tools that a design meets its requirements
 - Verifiability is not guaranteed by any of the FDTs

Selected Model Checking Tools

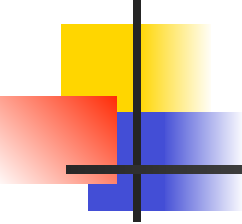
- Spin
 - Model checking tool where models are expressed in PROMELA (PROcess MEta Language)
 - Available with source code on Windows and Unix platforms
 - <http://spinroot.com/spin/whatispin.html>
- VeriSoft
 - Model checking tool where models are expressed in C/C++
 - Available on Linux or Solaris platforms
 - <http://cm.bell-labs.com/who/god/verisoft/>
- PV: The Protocol Verifier
 - Model checking using PROMELA
 - Available on Linux or Solaris platforms
 - http://www.cs.utah.edu/formal_verification/software/pv/
- Murphi
 - Custom description language
 - Available on Linux, Solaris, Irix, and HP-UX
 - <http://sprout.stanford.edu/dill/murphi.html>



Validation of Network Protocols

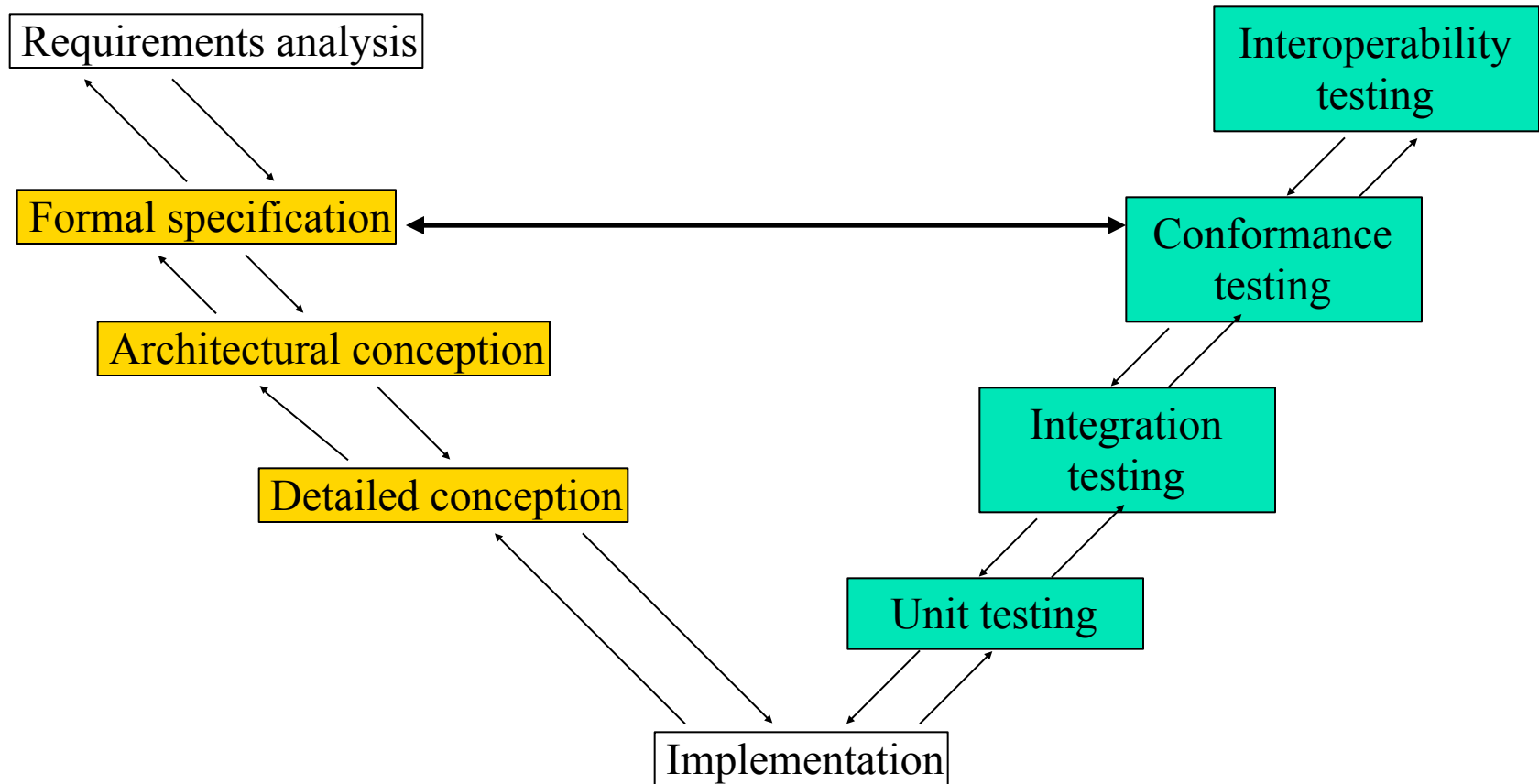
Dário Vieira

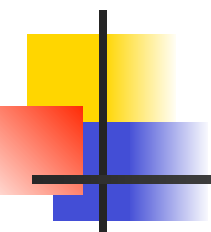
EFREI



Overview

Protocol life cycle



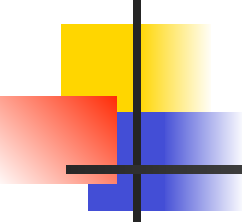


Verification vs. validation

- Verification: "Are we building the product right"
 - The software should conform to its specification
- Validation: "Are we building the right product"
 - The software should do what the user really requires
- V & V must be applied at each stage in the software process
- Two principal objectives
 - Discovery of defects in a system
 - Assessment of whether the system is usable in an operational situation

Formal Methods





Formal Methods Techniques

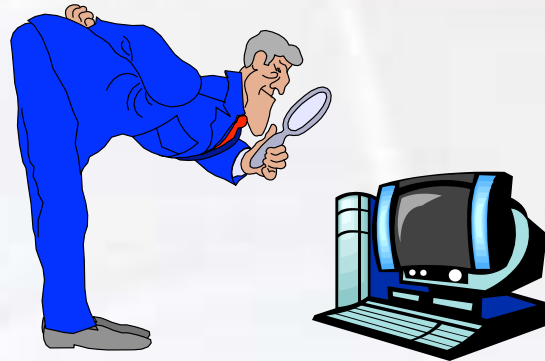


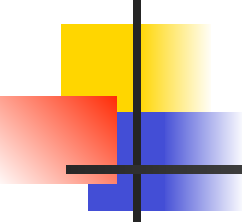
We will focus on

➤ **Two techniques**

Formal Verification

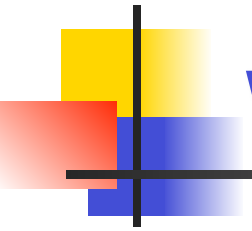
Formal Testing





What are Formal Methods?





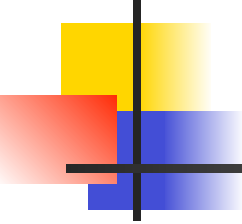
What are Formal Methods?

➤ Definition

- Mathematically based techniques for the specification, development and verification of software and hardware systems

➤ Can determine if

- ⇒ Specifications are satisfied
- ⇒ Implementation of a system is correct
- ⇒ Proving properties of a system



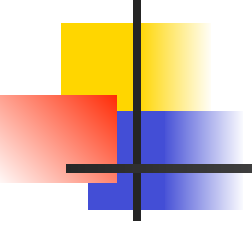
What are Formal Methods?

➤ Formal Language Specification

- Formal Methods centered around a notation known as a formal specification language
- **Increase Human Understanding of Specified System**
- **Allow the possibility of formal reasoning and development**
- **Examples: B (Wordsworth, 1996) and Z (Hayes, 1987) (Spivey, 1992)**

➤ Formal Semantics

- **Mathematical base allows precise notions**
- **Unambiguous**
- **Allows consistency, correctness, specification and implementation to be expressed**



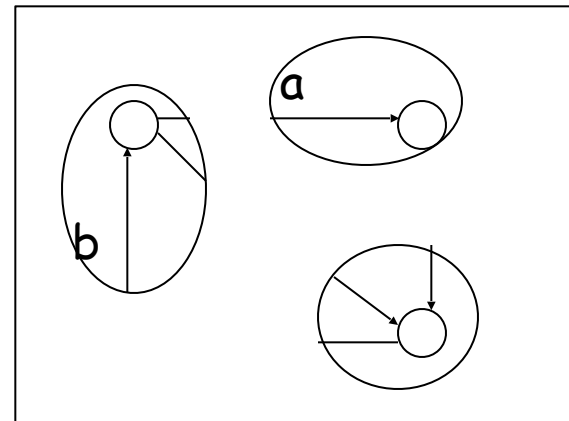
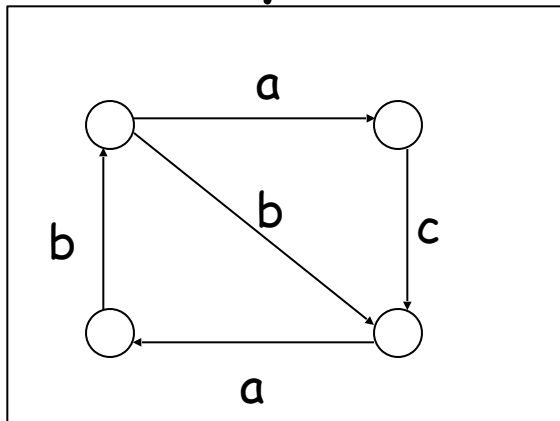
What are Formal Methods?

➤ Specification Approaches

- Approaches used to specify a model of a system using specification languages
 - **Model-Orientated:** Specify behavior in terms of mathematical structures
Examples: sets, sequences, maps, functions
 - **Property-Orientated: Specify necessary minimal constraints of the systems behavior**
Examples: Algebraic methods, first-order predicate logic

Testing problem

- A specification gives a description of the system behavior.
- An implementation should behave according to the specification.
- Is this really true?



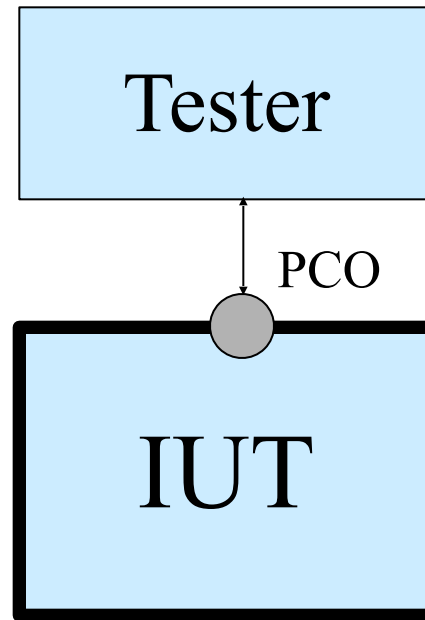
Black-box vs. White-box testing

- Black-box: **unknown** system structure
 - Testing against reference specification
 - Use the specification interface for
 - test selection, coverage criteria, result analysis
- White-box: **known** system structure
 - Testing against reference specification
 - Use system structure for
 - test selection, etc.

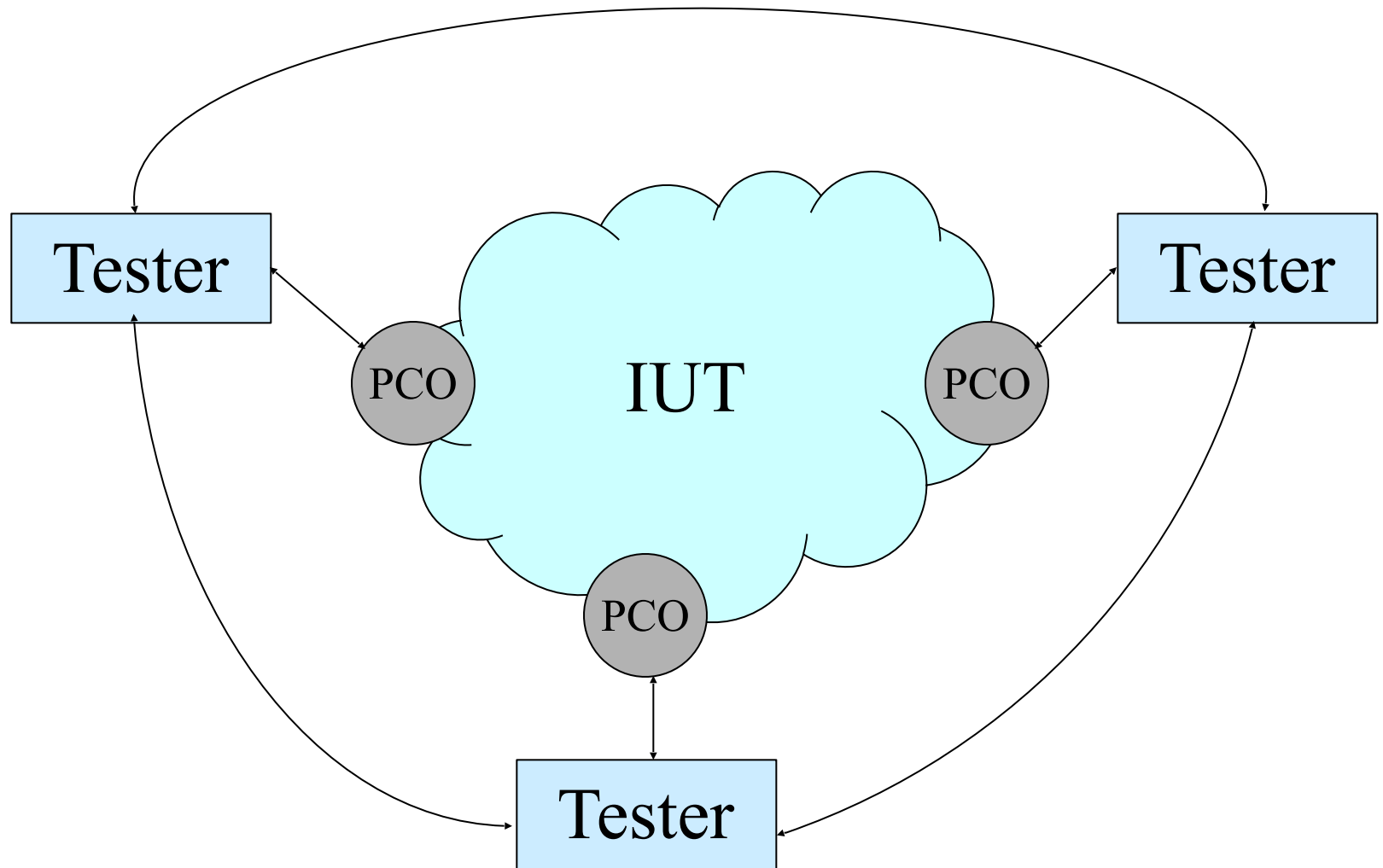
Protocol testing architectures

- IUT - implementation under test
- PCO - point of control and observation

Local
architecture



General testing architecture



Tests and test suites

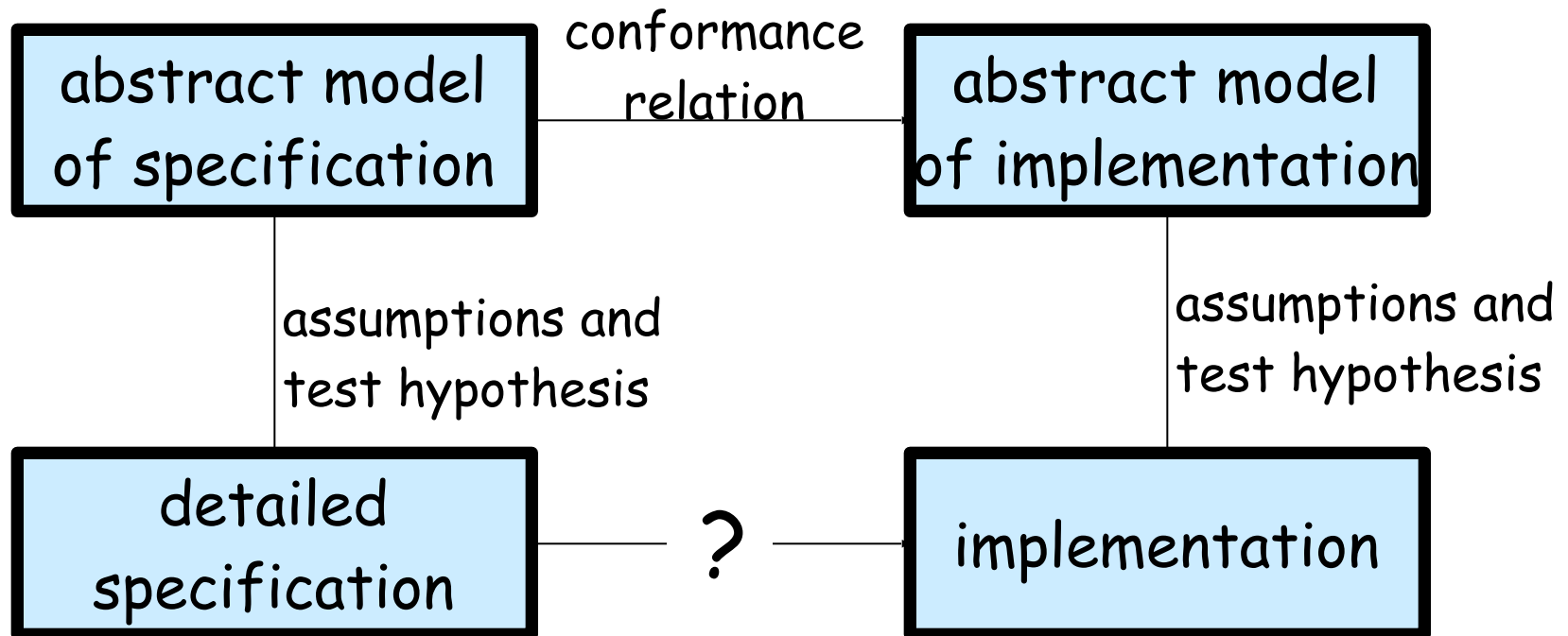
- A **test case** defines:
 - A finite sequence of inputs
 - A finite sequence of expected outputs
- A **test suite** is a collection of tests to achieve a certain **test coverage**
- Testing process
 - test generation: construct a test suite
 - test application: execute each test
 - result evaluation: interpret outputs

Test structure

- Test **purpose**
 - e.g., identify a state of the implementation
- Test **preamble**
 - lead the IUT into known state
- Test **body**
 - invoke behavior corresponding to the test purpose
- **Checking** of test results

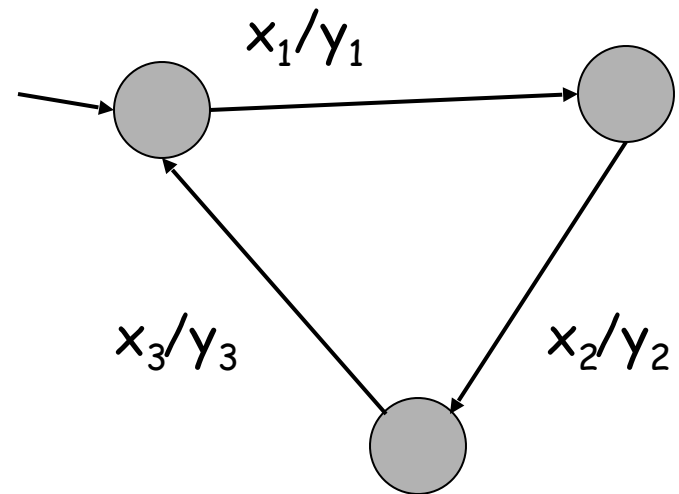
Modeling approach

- Testing criteria on a concrete system are hard to formulate
- Abstract models make reasoning easier



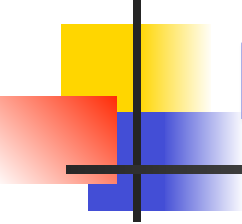
FSM model

- A set of states
- An initial state
- Finite sets of input (X) and output (Y) events
- Transfer function
- Output function



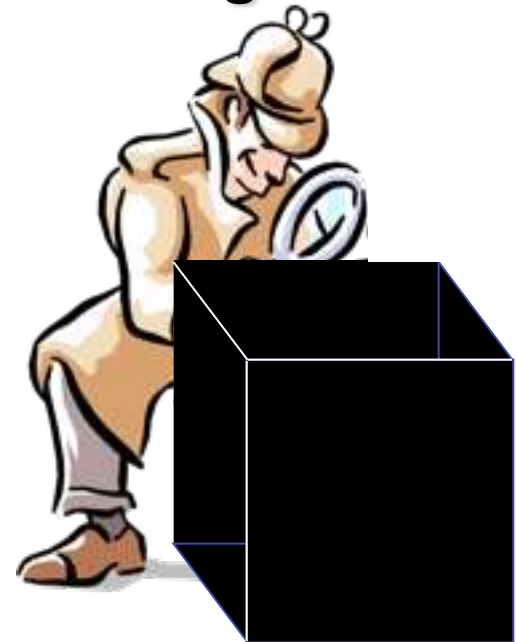
FSM-based fault model

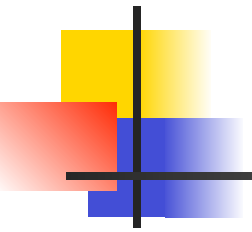
- Output faults
- Transfer faults
- Transfer faults with additional states
- Additional or missing transitions
- Control and data flow faults?



Finite State Machines

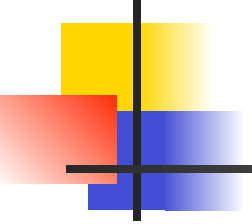
- **Many systems can be specified or modeled as state machines**
- **State machines as the basis for testing**
 - ⇒ FSM as basis for functional tests
 - ⇒ black box
 - ⇒ specification based
 - ⇒ reactive systems :
 - ✧ communication protocols
 - ✧ control systems
 - ✧ embedded systems





Finite State Machines

- A Finite State Machine consists of:
 - K states: $S = \{s_1, s_2, \dots, s_k\}$, s_1 is initial state
 - N inputs: $I = \{i_1, i_2, \dots, i_n\}$
 - M outputs: $O = \{o_1, o_2, \dots, o_m\}$
 - Next-state function $T(S, I)$ mapping each current state and input to next state
 - Output Function $P(S)$ specifies output



Formal Definition of a FSM

- A DFA is a five-tuple:

$$M = (Q, \Sigma, \delta, q_0, F)$$

Q A finite set of states

Σ A finite input alphabet

q_0 The initial/starting state, q_0 is in Q

F A set of final/accepting states, which is a subset of Q

δ A transition function, which is a total function from $Q \times \Sigma$ to Q

$\delta: (Q \times \Sigma) \rightarrow Q$ δ is defined for any q in Q and s in Σ , and
 $\delta(q, s) = q'$ is equal to some state q' in Q , could be $q' = q$

Intuitively, $\delta(q, s)$ is the state entered by M after reading symbol s while in state q .

Finite State Machines

➤ Components of FSM

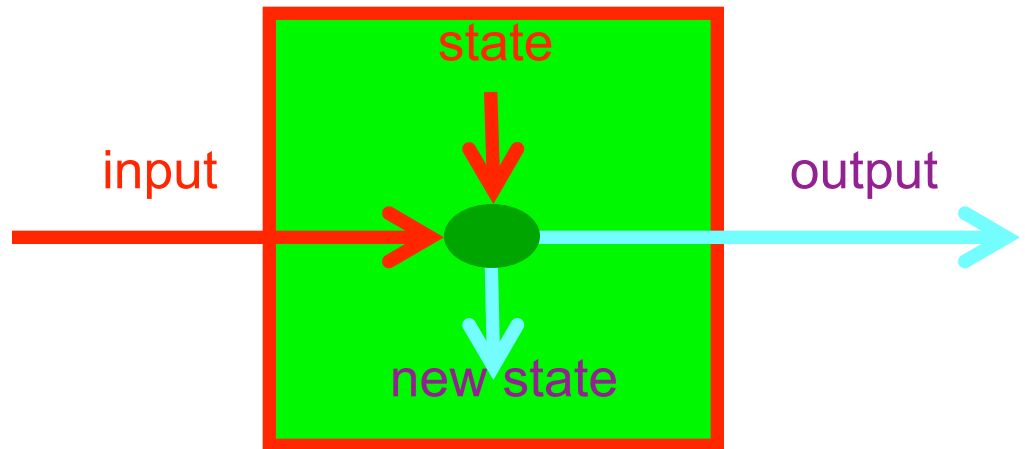
➤ States

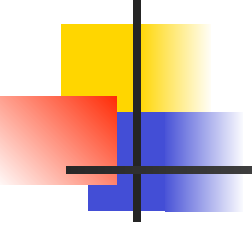
➤ Transitions

➤ Inputs (“triggers”)

➤ Outputs

➤ New state

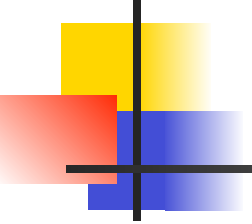




Representing a Finite State

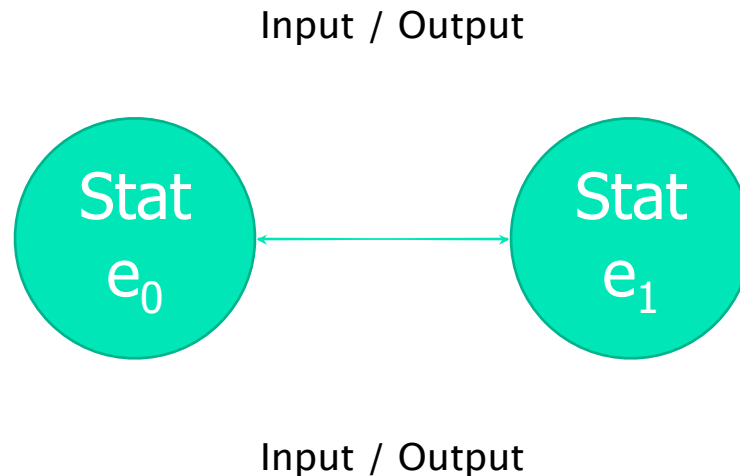
- It can be represented using a **state transition table** which shows the current state, input, any outputs, and the next state

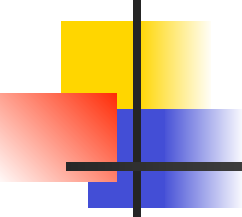
State	Input	Input		
	Current			
State		Next State / Output		Next State / Output
State				
....				
State				



Representing a Finite State Machine

- It can also be represented using a **state diagram** which has the same information as the state transition diagram

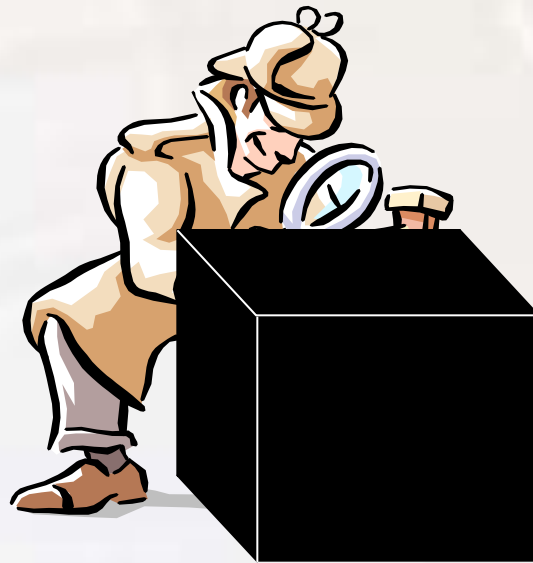




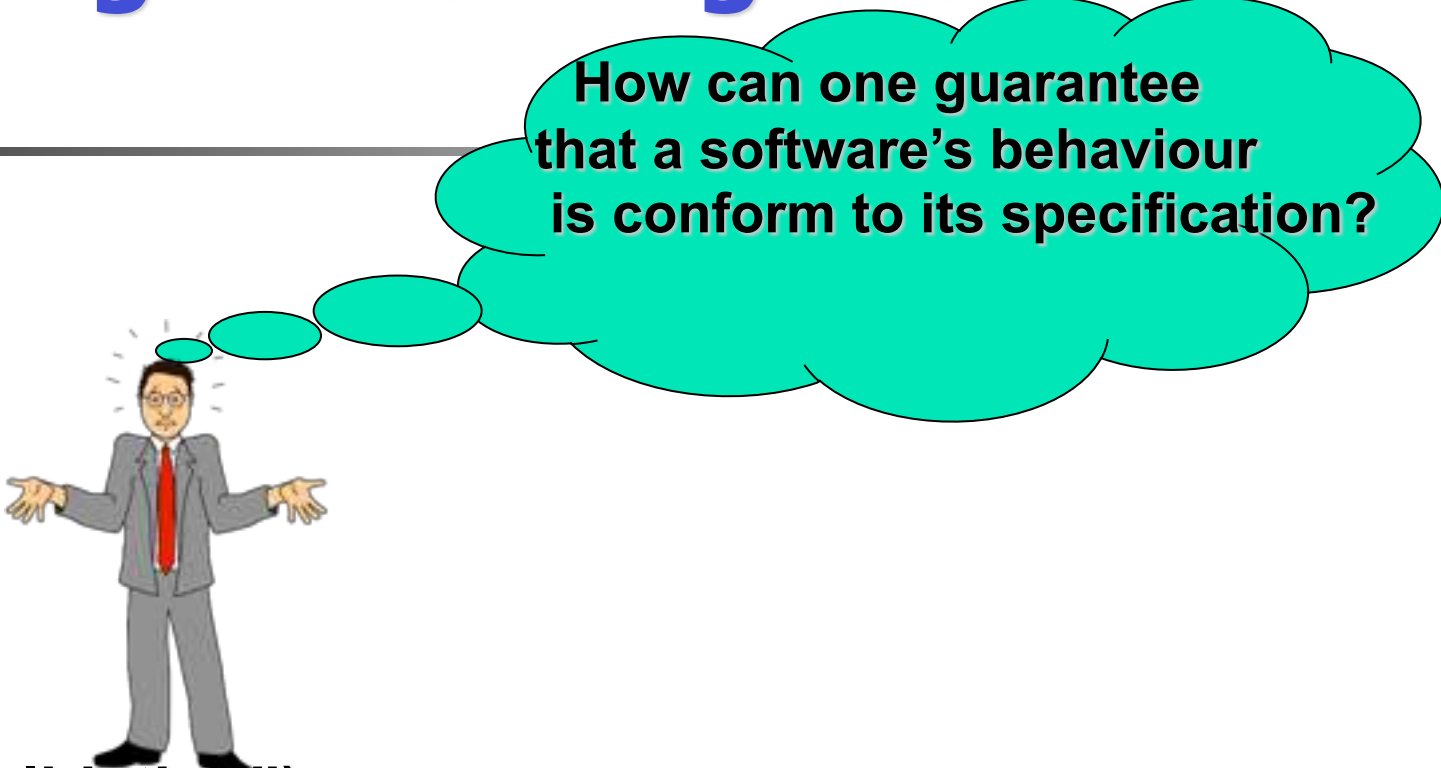
Finite State Machines

- The output function produces a set of outputs from the current state and the inputs
 - ↳ There are two types of finite state machines
 - ↳ In a **Moore machine**, the output only depends on the current state
 - ↳ While in a **Mealy machine**, the output depends both the current state and the current input
 - ↳ These **two types are equivalent** in capabilities

Formal Testing



Testing IP Routing Protocols



How can one guarantee that a software's behaviour is conform to its specification?

✧ Testing (“Validation”)

➤ Process of trying to find errors in a system by means of experimentation

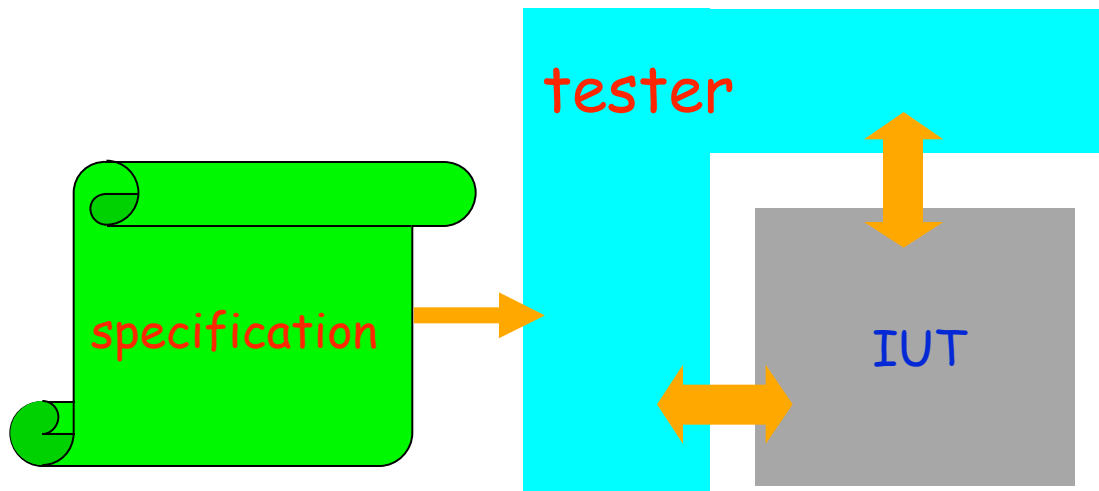
✧ Test architecture

➤ It is the description of the environment in which a system is tested

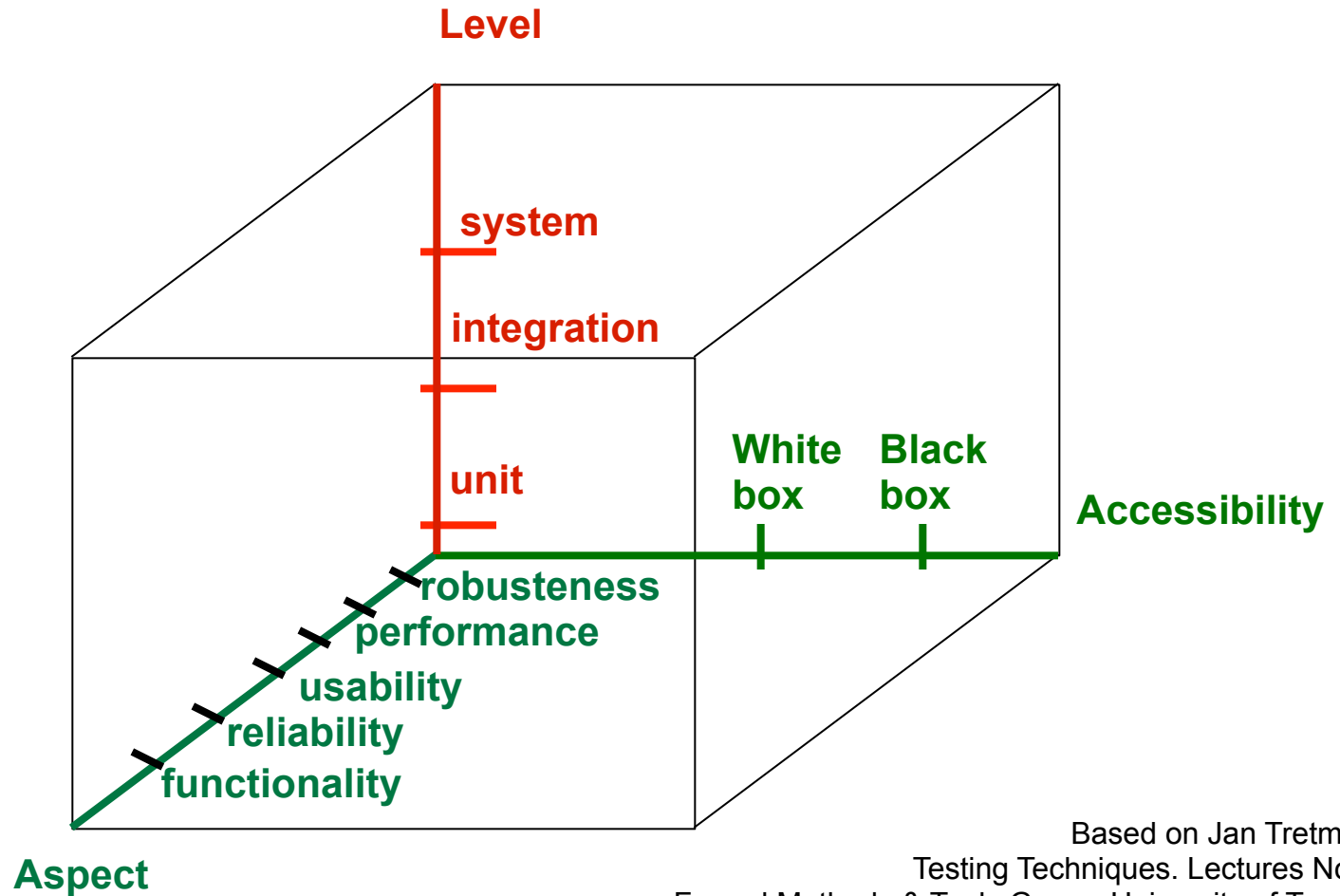
Testing

Testing

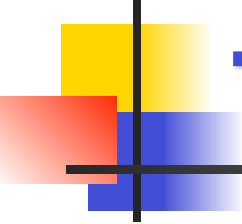
checking or measuring some quality characteristics of an executing object by performing experiments in a controlled way w.r.t. a specification



Types of testing



Based on Jan Tretmans
Testing Techniques. Lectures Notes
Formal Methods & Tools Group, University of Twente,
The Netherlands, 2001



Testing

➤ We can classify testing in two categories

➤ Black box testing

✧ Passive testing

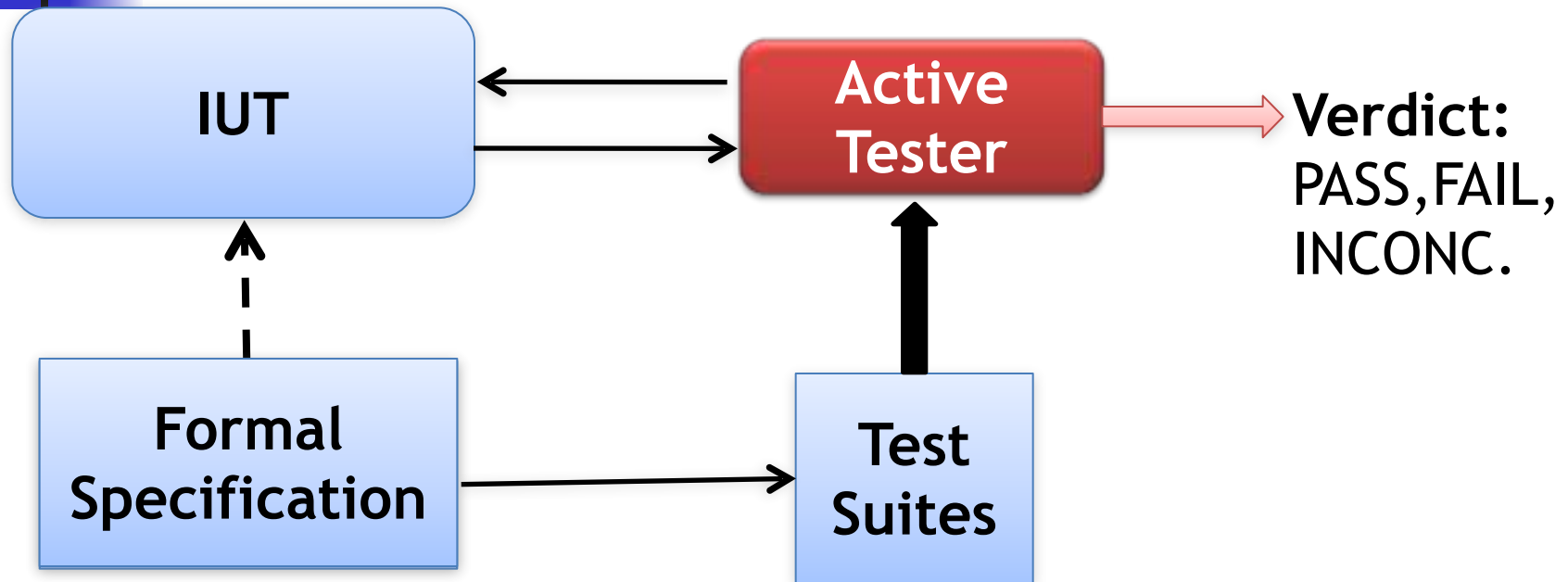
✧ Active testing

➤ White Box testing

Active Testing

pros et cons:

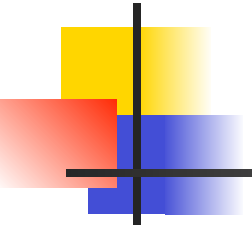
- possibility to focus on a particular area of the specification
- automatic test generation is difficult
- can disturb (or crash) the IUT



Automatic test generation based on formal descriptions

Generation of a : - reasonable test scenarios number (Execution)

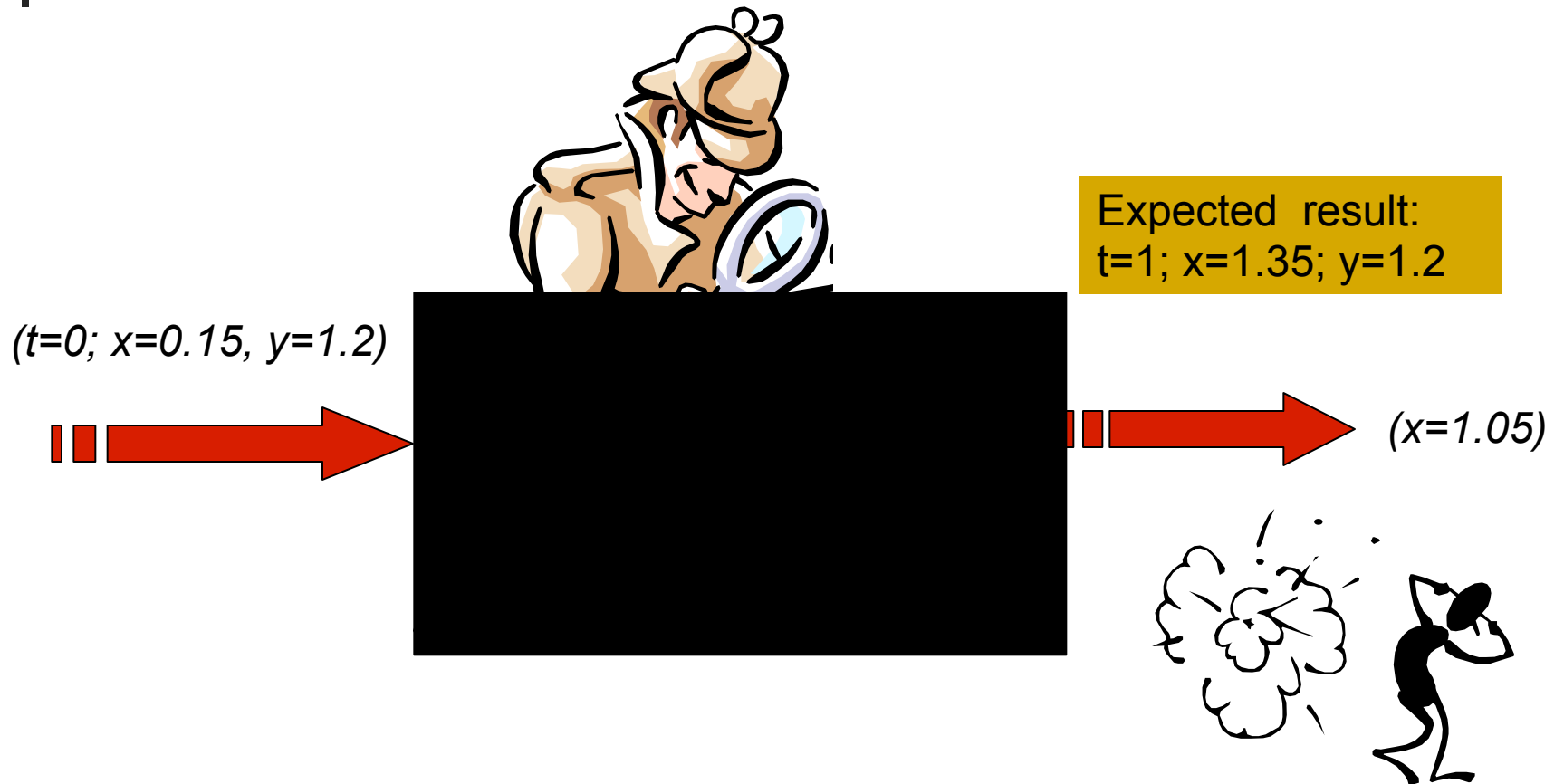
- Complete (to cover all the system transitions)



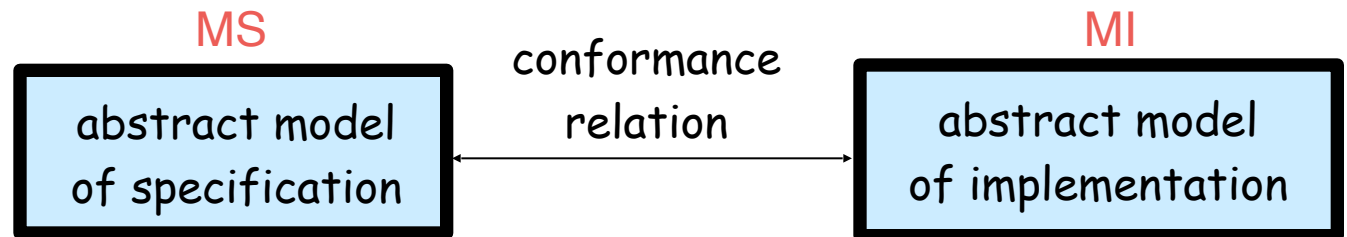
Black Box Testing

- Software tested to be treated as a **block box**
- Specification for the black box is given
- The expected behavior of the system is used to design test cases
- Test cases are determined solely from specification.
- Internal structure of code **not used** for test case design

Functional ("black box")



Testing of black box Finite State Machine



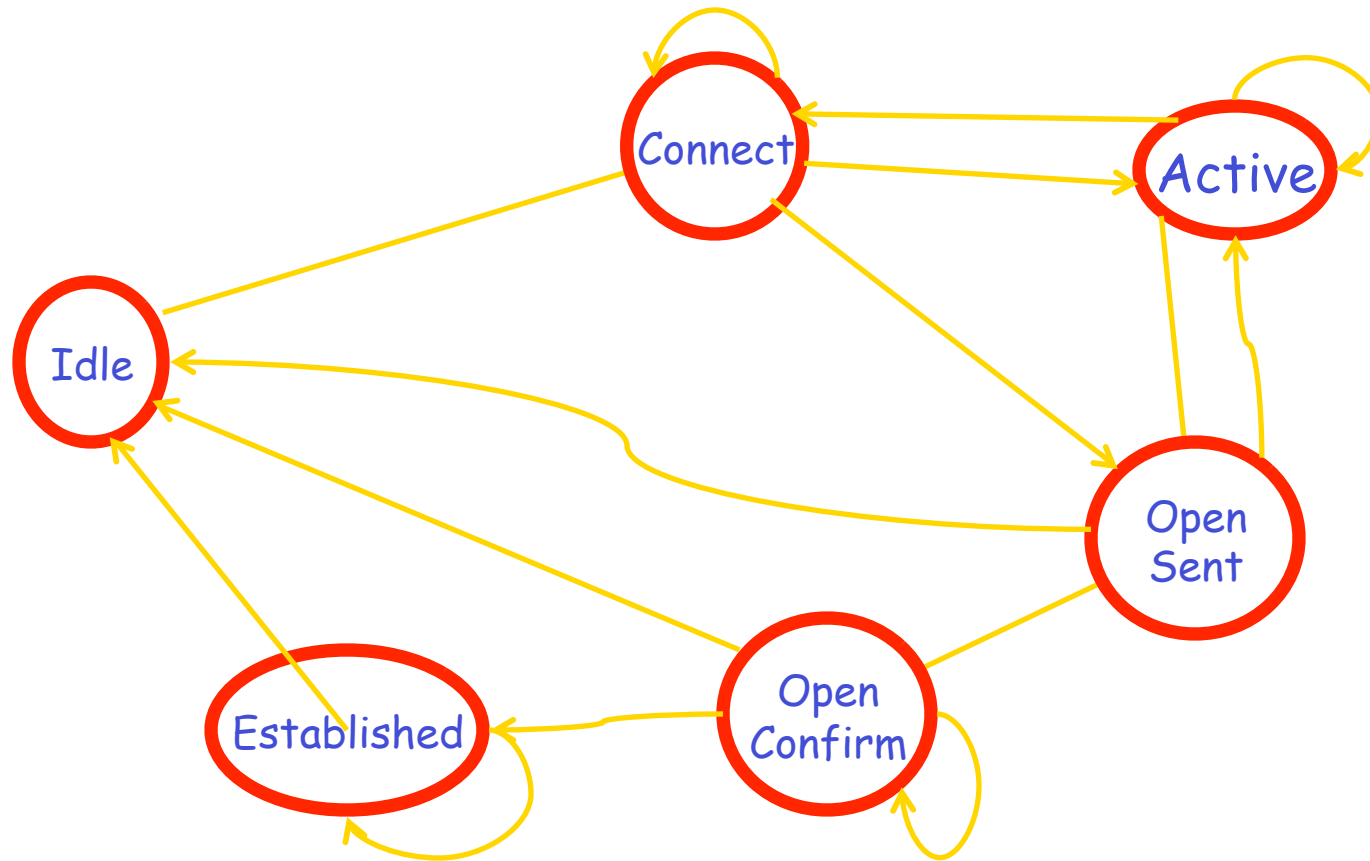
➤ Given

- a specification FSM **MS**
- a (black box) implementation FSM **MI**

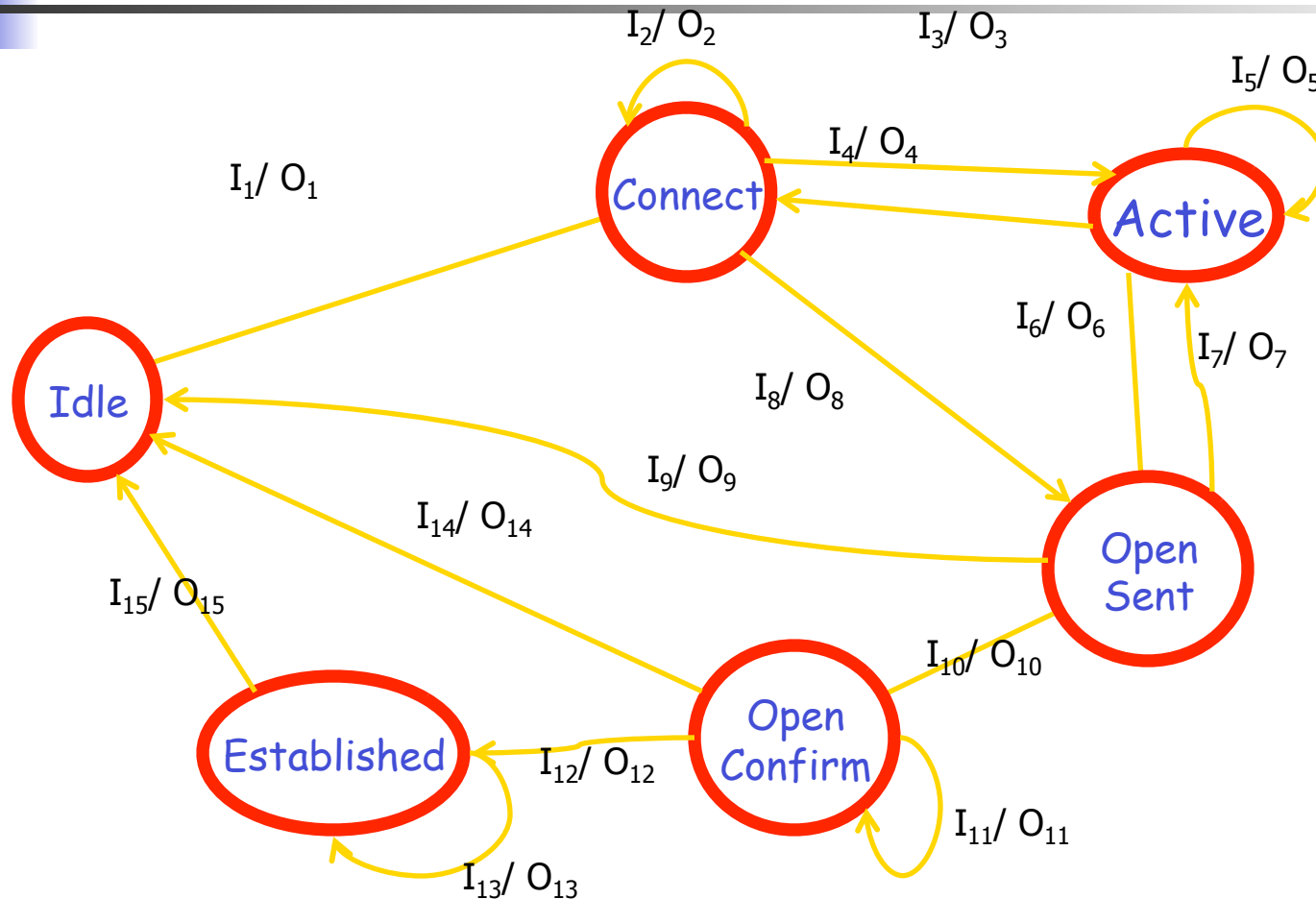
➤ Determine whether

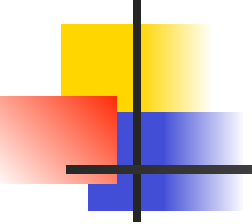
- **MI** conforms to **MS**.
- i.e., **MI** behaves in accordance with **MS**
- i.e., whether outputs of **MI** are the same as of **MS**
- i.e., whether the reduced **MI** is equivalent to **MS**

BGP FSM Model



BGP FSM Model





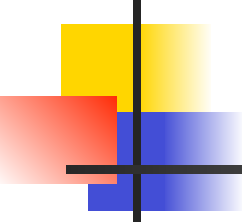
Testing of black box Finite State Machine

➤ Possible errors

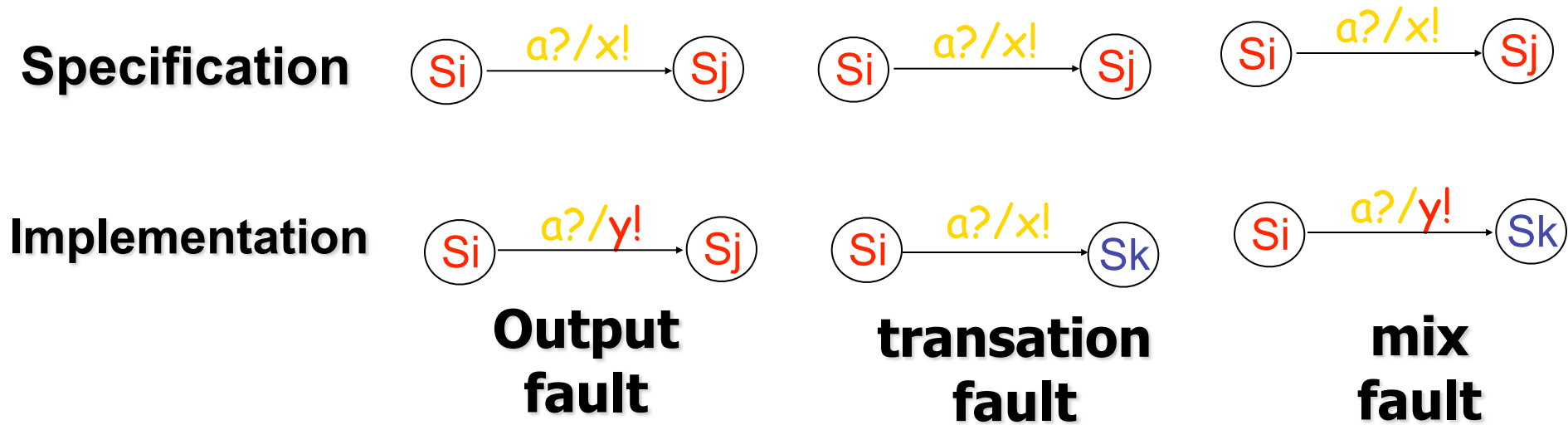
- ✚ extra or missing states
- ✚ output fault
- ✚ transition fault
 - ✧ to other state
 - ✧ to new state

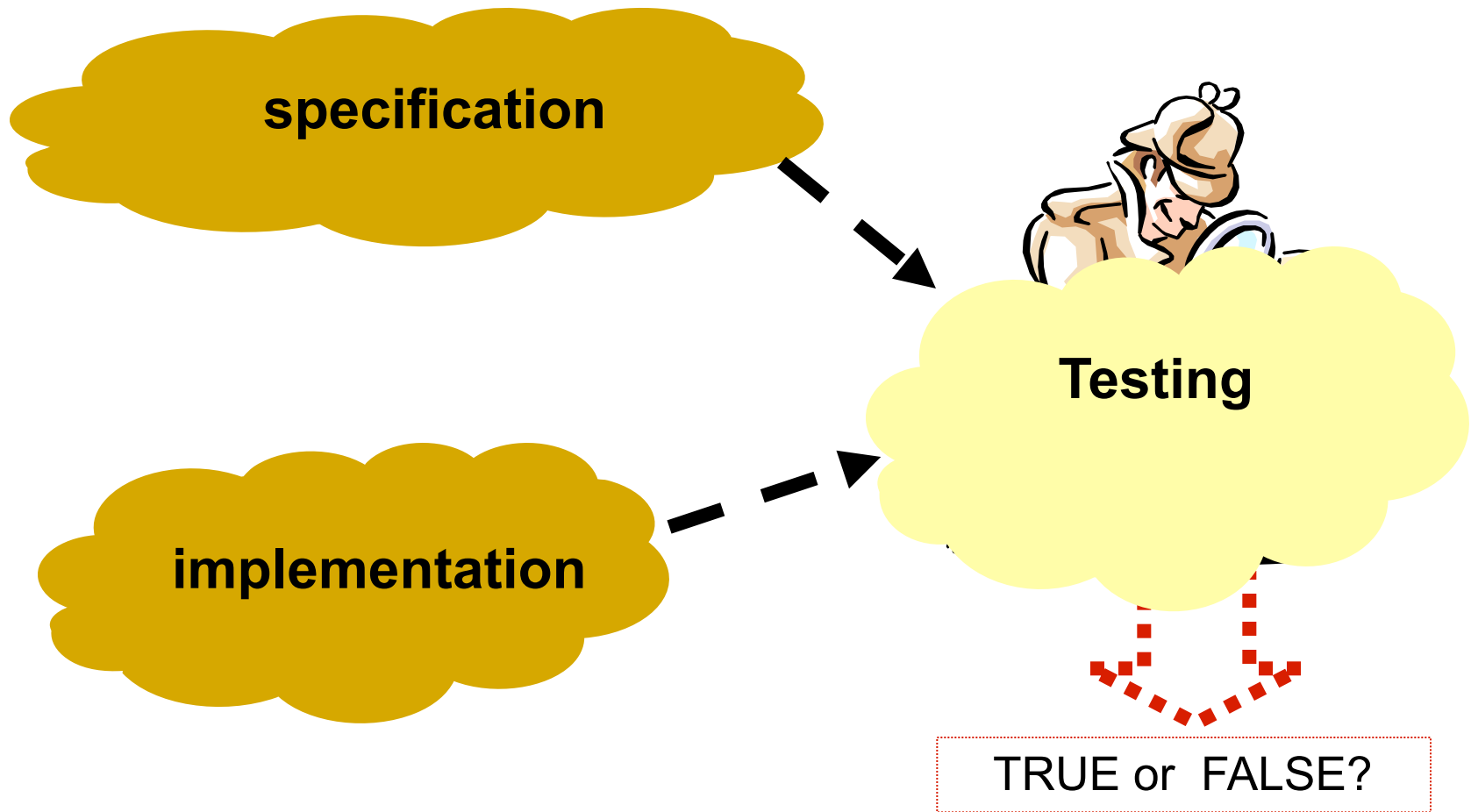
➤ What should be tested?

- ✧ the path (sequence of inputs with expected outputs) and
- ✧ transitions



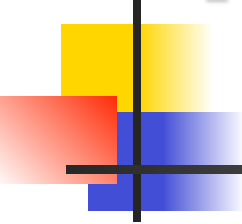
Testing of black box Finite State Machine





■ Conformance Test

- ❑ To verify if an implementation is according to the specification



Testing of black box finite state machine

- Test with paths of the (specification) FSM
- Path = sequence of inputs with expected outputs
(cf. path testing as white-box technique)
- Infinitely many paths : how to select ?
- Different strategies :
 - ↳ test every state : state coverage (of specification !)
 - ↳ test every transition : transition coverage
 - ✧ test output of every transition
 - ✧ test output + resulting state of every transition



Model Construction

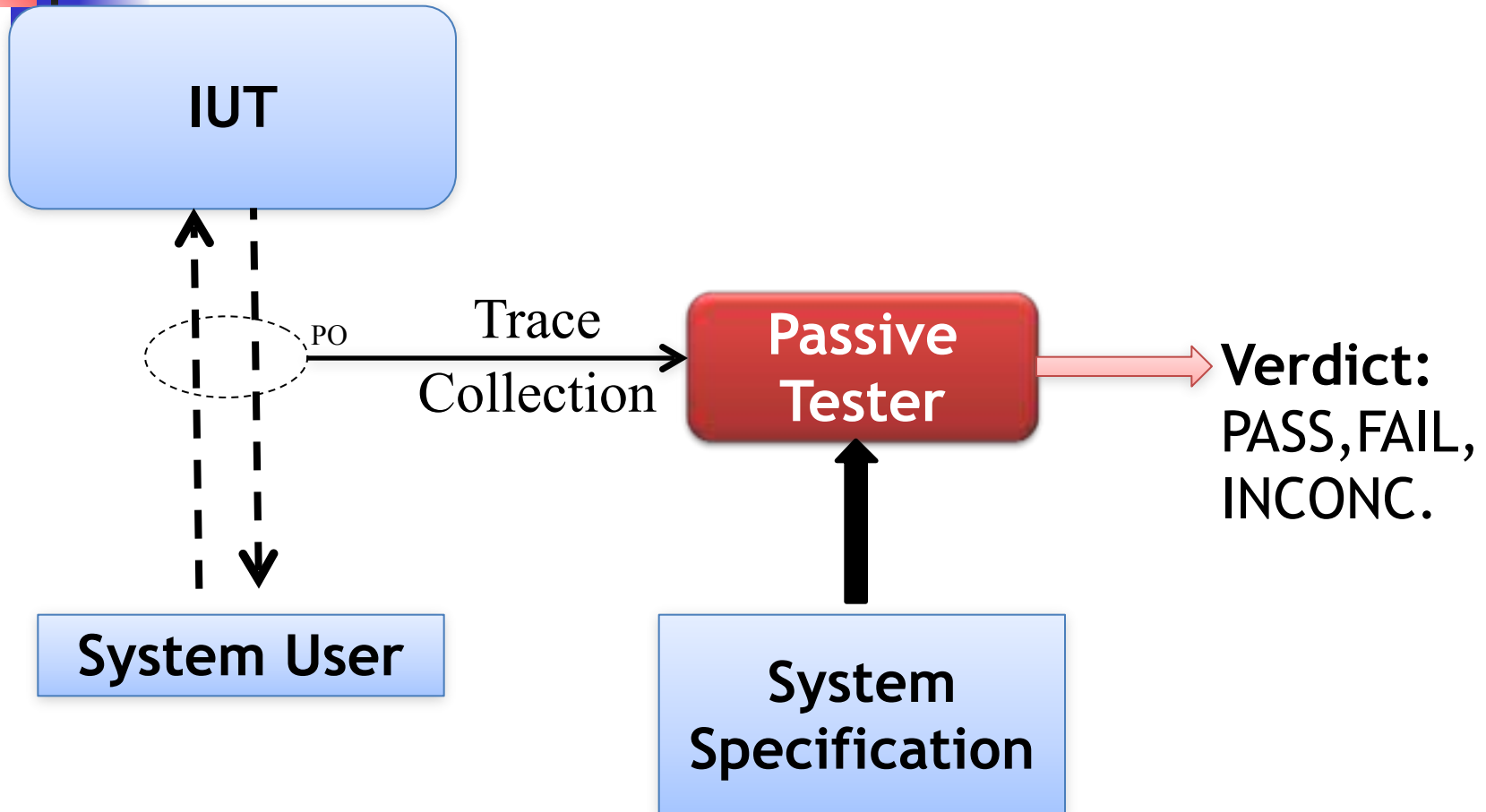
What's the obstacle?

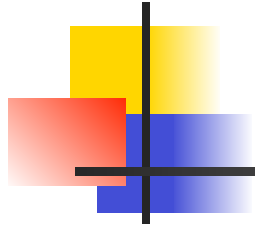


Passive Testing

pros et cons:

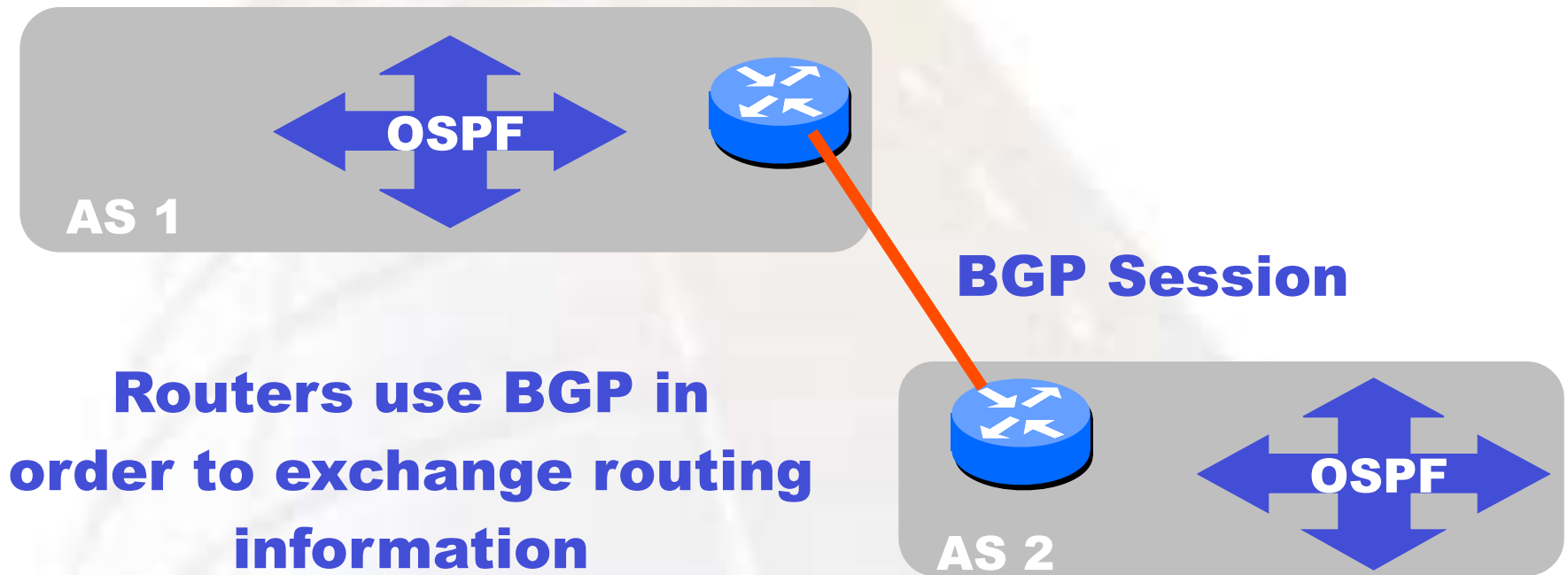
- no interference with the IUT
- no need of tests generation
- algorithms efficiency



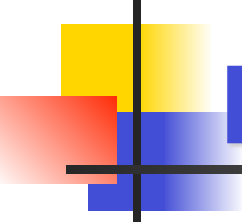


Features of Routing Protocols

Architecture of Dynamic Routing



**The Routing Domain of BGP is
the entire Internet**



Desired Properties

2 Some Desired Properties

- Ø Correctness

- Ø Robustness and stability

 - ✧ A network is expected to run continuously for many years, with changes in topology and traffic

 - ✧ The routing algorithm will converge after a disturbance

- Ø Flexibility

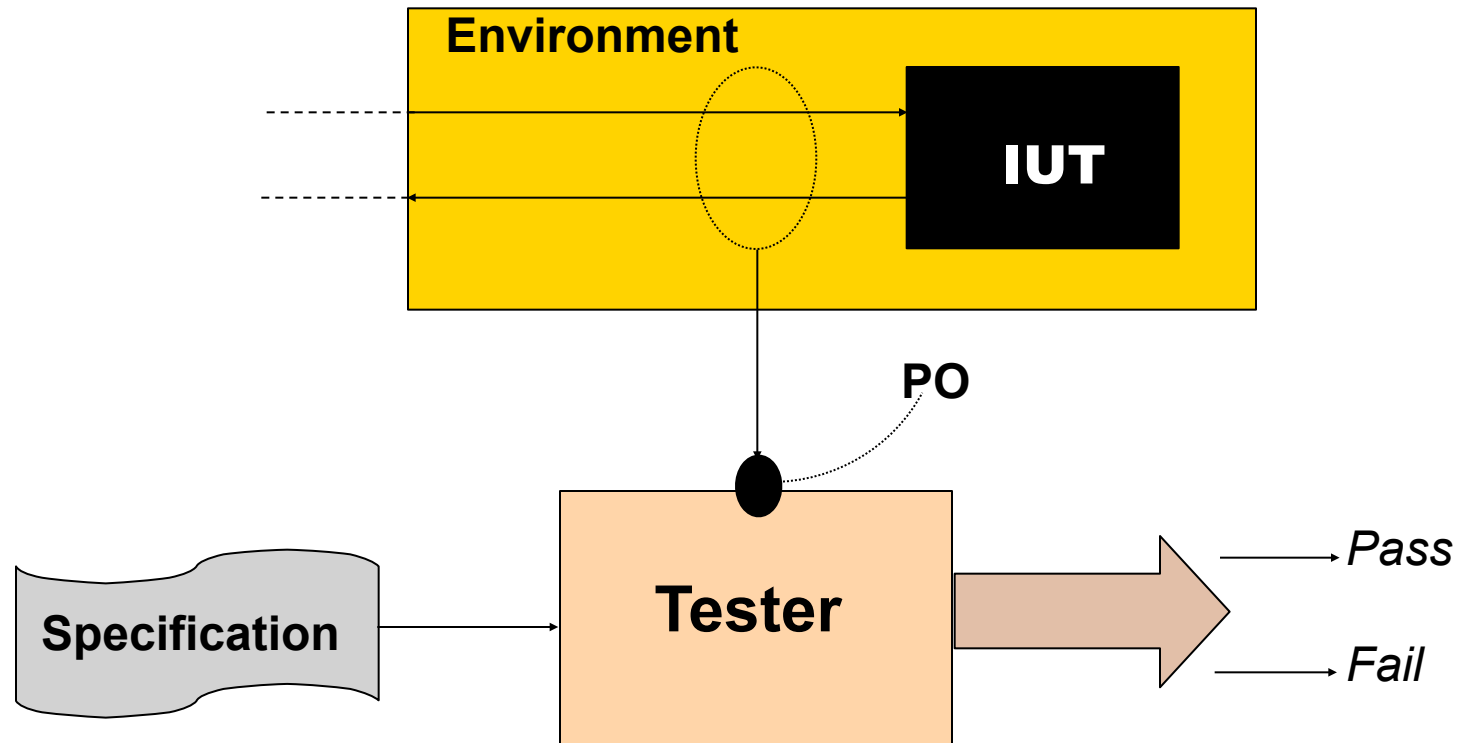
 - ✧ routing protocols should quickly and accurately adapt to many network circumstances

 - ✧ network delay, router queue size, changes in network bandwidth

- Ø Optimality

 - ✧ routing protocol should be able to choose the best route to send a packet according to predefined metrics

Passive Testing Architecture



Passive Testing Architecture

$i_1/o_1, \dots, i_{n-1}/o_{n-1}, i_n/O$

Translation into an appropriate form

$(s_0, i_1/o_1, s_1)$
 $(s_1, i_2/o_2, s_2)$
 $\vdots$
Aldebaran form

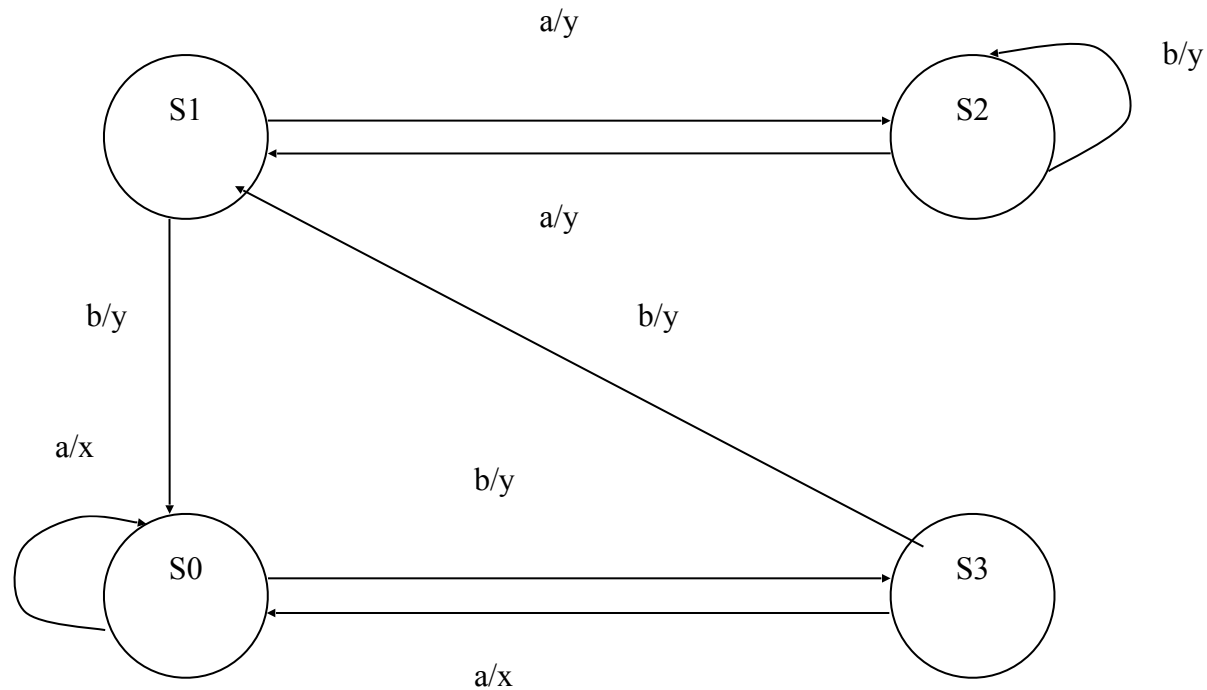
Invariant

pass

Algorithm for
Pattern matching
on string

fail

Example/Exercise

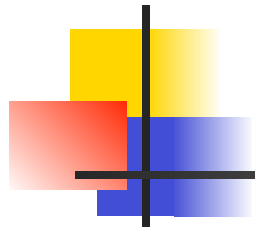


a/x b/y a/x b/y b/y a/y b/y a/y b/y (trace 1)

a/x b/y a/x b/y b/y a/x b/x a/y b/y (trace 2)

Formal Verification





Formal Verification

- **Definition**

- ✓ It is used to guarantee that the system specification is correct.

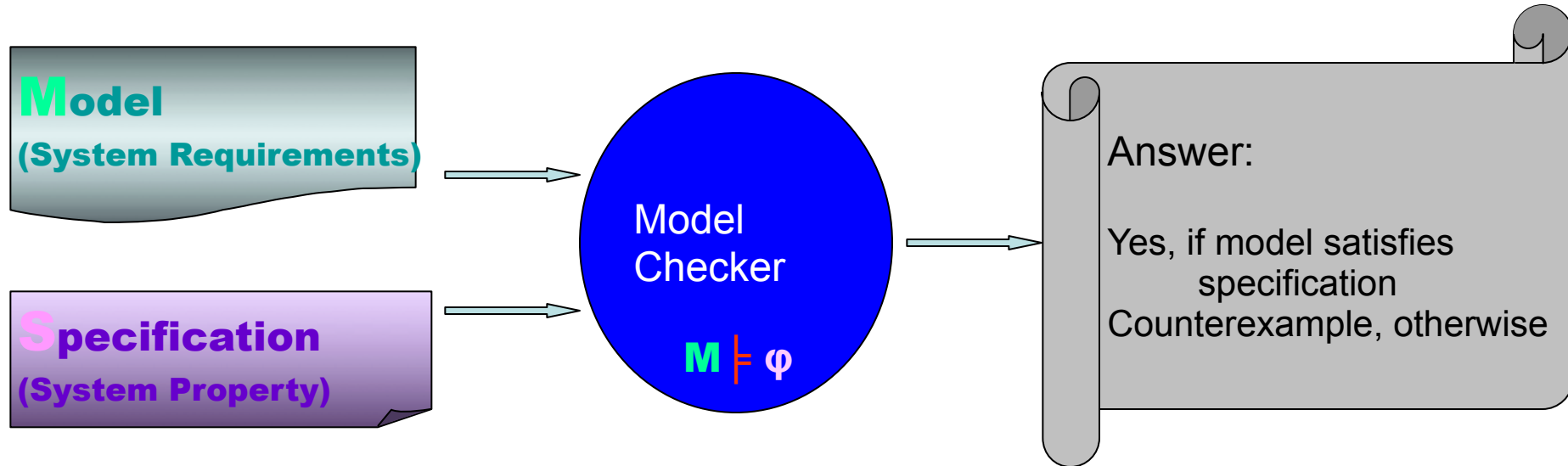
- **There are many approaches**

- ✓ Automated theorem proving
- ☞ Model Checking



Model Checking Process

[Adapted from www.lix.polytechnique.fr/comete/seminar/1-ModelChecking.ppt]



For increasing our confidence in the correctness of the model:

- ☐ Verification: The model satisfies important system properties
- ☐ Debugging: Study counter-examples, pinpoint the source of the error, correct the model, and try again

Model Checking

- **Definition**

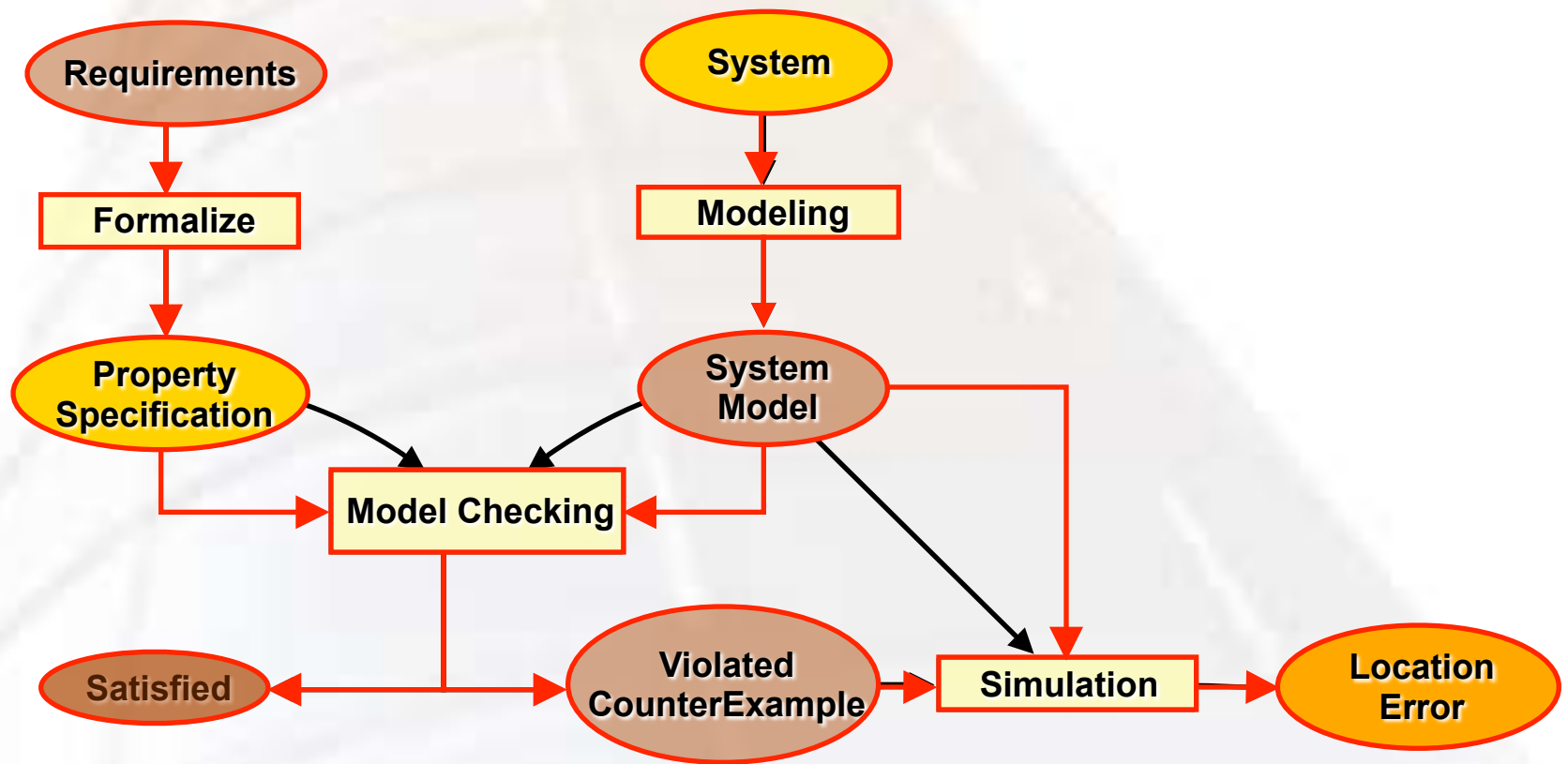
- It is a verification technique that explore all possible system states in an exhaustive way

- **Two main advantages:**

- Its application requires no user supervision or expertise in mathematical disciplines
- When the design fails to satisfy a desired property, we have always a counterexample

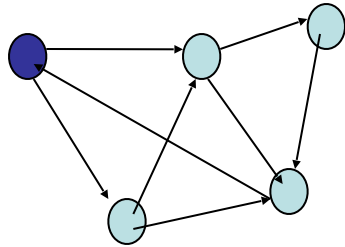


The Model Checking Process



Model Checking

state diagram



property

Φ

model checker:
automatic,
exhaustive search

Yes or No & diagnostic trace

Model checking, more philosophically

in the end:

the verification result is only as good as the model that is used for verification.

minimalistic point of view:

model checking is only a debugging technique:

every error we find by model checking
& that can be traced back in the original system, is an error.

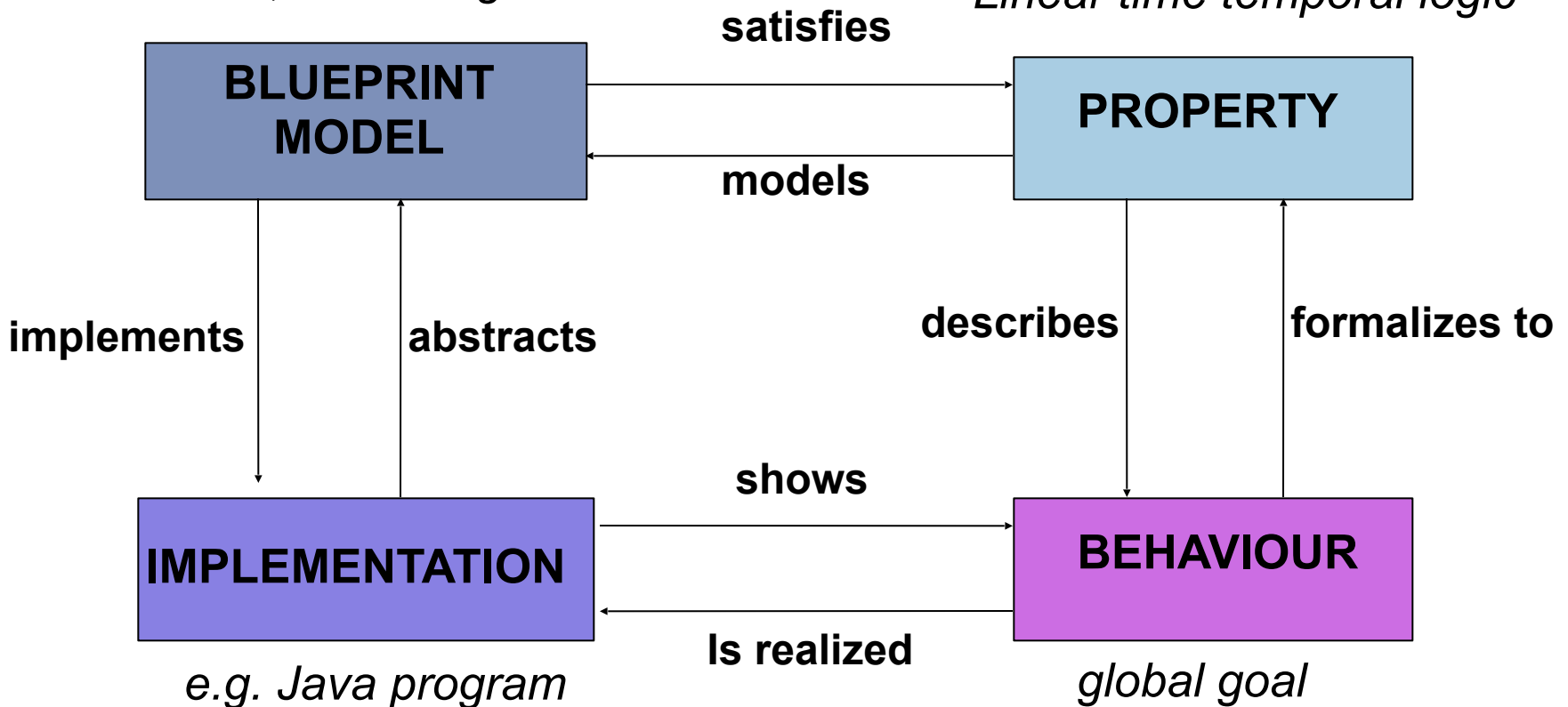
if we did not find errors, we still do not know, whether the errors were abstracted away when modelling.

way out: systematic modelling,
make the modelling process explicit

Taxonomy of concurrent program correctness

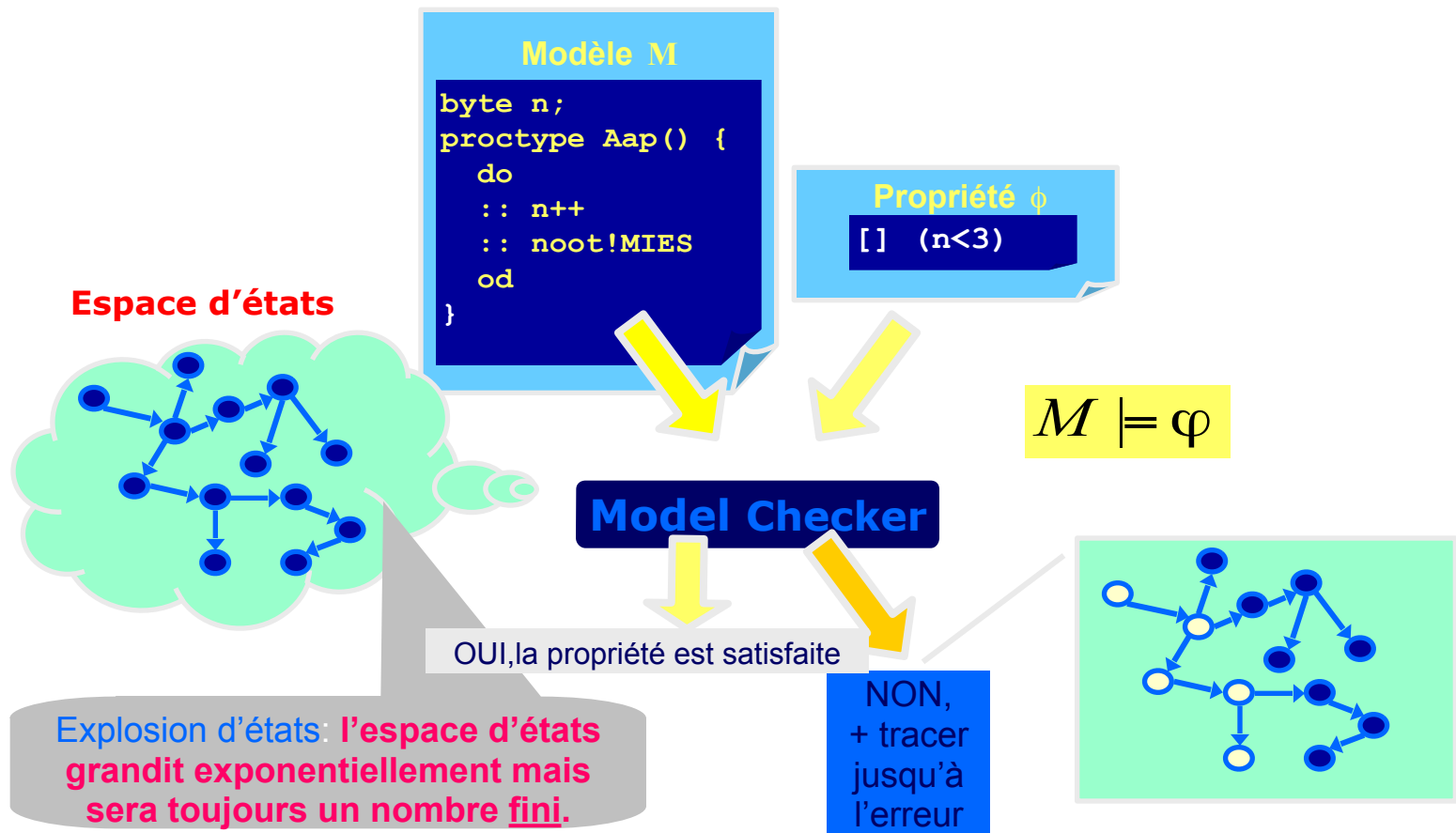
*In a formal language:
Promela, state diagrams...*

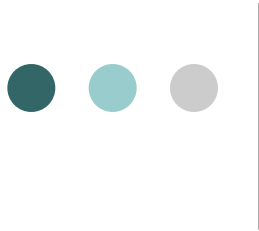
*In a formal language:
Linear time temporal logic*



Analyse des modèles , Model Checking

- Est-ce que le modèle M vérifie la propriété ϕ dans tous les cas après l'explosion d'états : oui ou non, si non, tracer l'erreur.



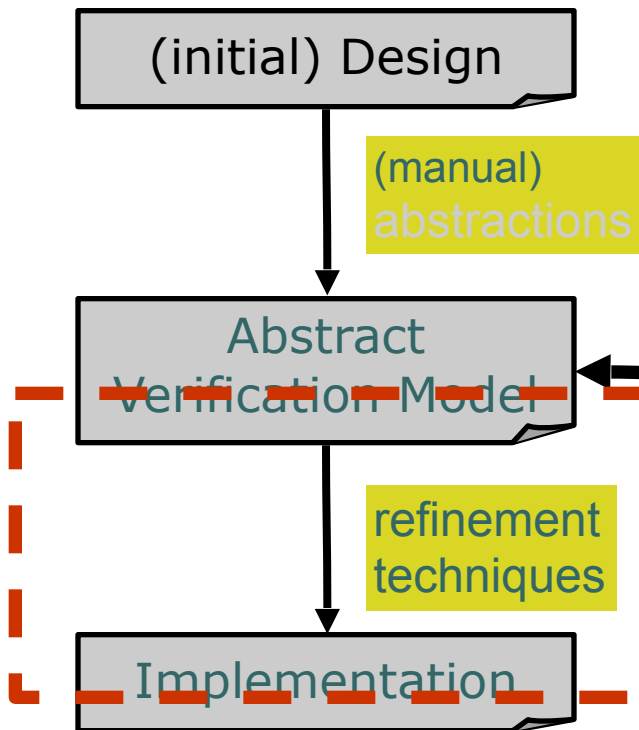


Analyse moderne et analyse classique

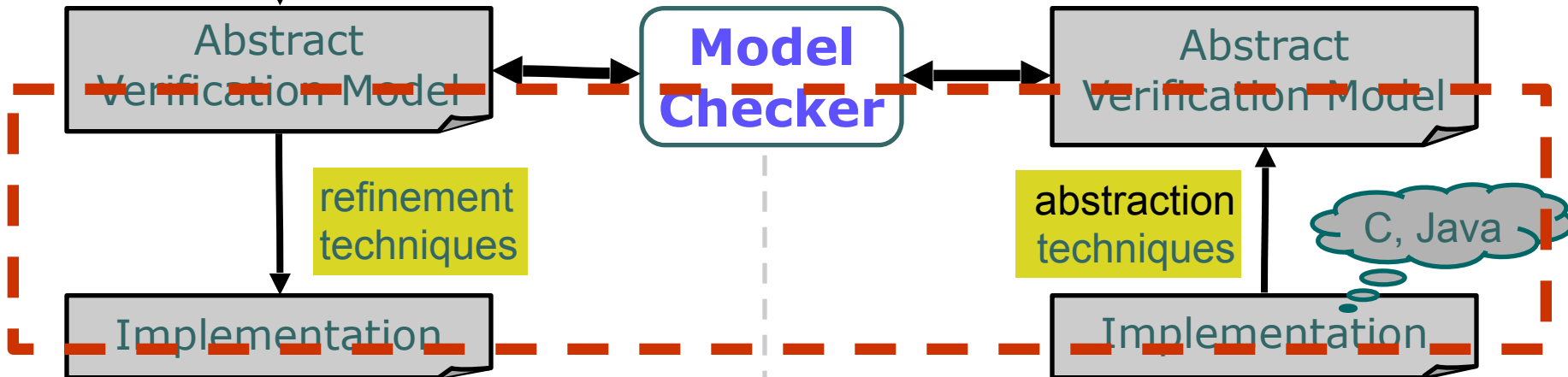
- Dans l'industrie informatique on distingue deux approches différentes d'analyse de modèles :
 - **Approche traditionnelle :**
 - Dans le processus de conception, on crée un modèle abstrait du processus avant de commencer l'implémentation.
 - L'analyse est faite sur ce modèle.
 - L'implémentation est obtenue à partir du modèle par un procédé de raffinement.
 - **Approche dite 'moderne' :**
 - Le modèle est obtenu à partir de l'implémentation par un procédé d'abstraction qui est automatisable.

Approches Classique et “Moderne”

Classic Approach



Modern Approach



Abstraction is the key activity in both approaches.

To cope with the state space explosion.

Basic Properties

- Safety properties
 - Invariants, deadlocks, reachability, etc.
 - Can be checked on finite traces
 - “something bad never happens”
- Liveness Properties
 - Fairness, response, etc.
 - Infinite traces
 - “something good will eventually happen”

Example: Mutual exclusion

It cannot happen that both processes are in their critical sections simultaneously.

Example: Mutual exclusion

It cannot happen that both processes are in their critical sections simultaneously.

Safety

Example: Bounded overtaking

Whenever process P_1 wants to enter the critical section, then process P_2 gets to enter at most once before process P_1 gets to enter.

Example: Bounded overtaking

Whenever process P_1 wants to enter the critical section, then process P_2 gets to enter at most once before process P_1 gets to enter.

Safety

Example: Starvation freedom

Whenever process P_1 wants to enter the critical section, provided process P_2 never stays in the critical section forever, P_1 gets to enter eventually.

Example: Starvation freedom

Whenever process P_1 wants to enter the critical section, provided process P_2 never stays in the critical section forever, P_1 gets to enter eventually.

Liveness

Example: Starvation freedom

Whenever process P1 wants to enter the critical section, provided process P2 never stays in the critical section forever, P1 gets to enter eventually.

Liveness

Model checking with SPIN

Safety properties, reachability properties:

algorithm traverses the state space

Liveness properties, evaluated on **infinite** computations

algorithm for nonemptiness of the intersection
of two automata (languages)

**message: choose different algorithms for safety
and liveness properties!**

Model Checking with SPIN

1. Concepts of the modelling language PROMELA
2. How to prove properties
2. Examples

Promela: Model

A Promela model consists of:

type declarations

mtype, constants,
typedefs (records)

channel declarations

chan **ch** = [**dim**] of {**type**, ...}
asynchronous: **dim** > 0
rendez-vous: **dim** == 0

global variable declarations

- **simple** vars
- **structured** vars
- vars can be accessed by **all** processes

process declarations

behaviour of the processes:
local variables + statements

[**init** process]

initialises variables and
starts processes

Promela: Model

- Promela model consist of:
 - type declarations
 - channel declarations
 - variable declarations
 - process declarations
 - [`init` process]
- A Promela model corresponds with a (usually **very large**, but) **finite transition system**, so
 - no unbounded **data**
 - no unbounded **channels**
 - no unbounded **processes**
 - no unbounded **process creation**

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

proctype Sender() {
    ...
}
proctype Receiver() {
    ...
}

init {
    ...
}
```

process body

creates processes

Promela: Processes

- A **process**
 - is defined by a **proctype** definition
 - executes **concurrently** with all other processes, independently of speed or behaviour
 - **communicates** with other processes
 - using **global** (shared) **variables**
 - using **channels**
- There may be **several processes** of the **same type**.
- Each process has its own **local state**:
 - **process counter** (location within the **proctype**)
 - contents of the **local variables**

Promela: Processes

- A **process type** (**proctype**) consist of
 - a **name**
 - a list of **formal parameters**
 - **local variable** declarations
 - **body**

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

The body consist of a sequence of **statements**.

Promela: Processes

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

```
proctype Foo(byte x) {  
    ...  
}  
  
init {  
    int pid2 = run Foo(2);  
    run Foo(27);  
}  
  
active[3] proctype Bar() {  
    ...  
}
```

number of procs. (opt.)

parameters will be initialised to 0

Promela : Variables and Types

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

Basic types

```
bit    turn=1;    [0..1]
bool   flag;      [0..1]
byte   counter;   [0..255]
short  s;          [-215.. 215 - 1]
int    msg;        [-231.. 231 - 1]
```

Arrays

```
byte a[27];
bit  flags[4];
```

array indexing
start at 0

Typedef (records)

```
typedef Record {
    short f1;
    byte  f2;
}
Record rr;
rr.f1 = ..
```

variable
declaration

Promela: Variables and Types

- Variables should be **declared**.
- Variables can be **given a value** by:
 - Initialisation
 - assignment
 - argument passing
 - message passing
(see **communication**)
- Variables can be used in **expressions**.

Most **arithmetic**, **relational**, and **logical** operators of C/Java are supported, including **bitshift** operators.

```
int ii;  
bit bb;
```

```
bb=1;  
ii=2;
```

assignment =

```
short s=-1;
```

declaration &
initialisation

```
typedef Foo {  
    bit bb;  
    int ii;  
};
```

```
Foo f;  
f.bb = 0;  
f.ii = -2;
```

equality test ==

```
ii*s+27 == 23;  
printf("value: %d", s*s);
```

Promela : Statements

- The body of a process consists of a **sequence of statements**.
A statement is either
 - **executable**: the statement can be executed **immediately**.
 - **blocked**: the statement **cannot** be executed.
- An **assignment** is **always executable**.
- An **expression** is also a statement; it is **executable** if it evaluates to **non-zero**.

executable/blocked
depends on the **global state** of the system.

$2 < 3$

always executable

$x < 27$

only executable if value of x is smaller **27**

$3 + x$

executable if x is not equal to **-3**

Promela : Statements

- The **skip** statement is **always executable**.
 - “does nothing”, only changes process’ process counter
- A **run** statement is **only executable** if a new process can be created (remember: the number of processes is bounded).
- A **printf** statement is **always executable** (but is not evaluated during verification, of course).

```
int x;  
proctype Aap()  
{  
    int y=1;  
    skip;  
    run Noot();  
    x=2;  
    x>2 && y==1;  
    skip;  
}
```

Executable if **Noot** can
be created...

Can only become
executable if **some other
process** makes **x** greater
than **2**.

Promela: Statements

- **assert** (<expr>) ;
 - The **assert**-statement is **always executable**.
 - If <expr> evaluates to zero, SPIN will exit with an **error**, as the <expr> “**has been violated**”.
 - The **assert**-statement is often used within Promela models, to check whether certain **properties are valid** in a state.

```
proctype monitor() {  
    assert(n <= 3);  
}  
  
proctype receiver() {  
    ...  
    toReceiver ? msg;  
    assert(msg != ERROR);  
    ...  
}
```

Promela: Statements

```
if
:: choice1 -> stat1.1; stat1.2; stat1.3; ...
:: choice2 -> stat2.1; stat2.2; stat2.3; ...
:: ...
:: choicen -> statn.1; statn.2; statn.3; ...
fi;
```

- If there is at least one **choice_i** (guard) executable, the **if**-statement is executable and SPIN non-deterministically chooses one of the executable choices.
- If no **choice_i** is executable, the **if**-statement is blocked.
- The operator “**->**” is equivalent to “**;**”. By convention, it is used within **if**-statements to separate the guards from the statements that follow the guards.

Promela: Statements

```
if
:: (n % 2 != 0) -> n=1
:: (n >= 0)      -> n=n-2
:: (n % 3 == 0) -> n=3
:: else         -> skip
fi
```

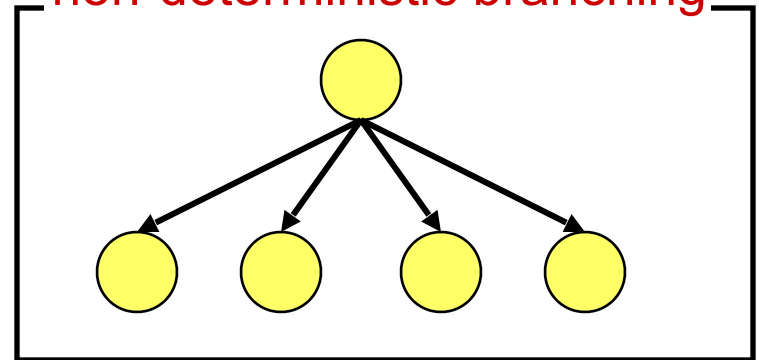
- The **else** guard becomes **executable** if **none** of the other guards is executable.

give n a random value

```
if
:: skip -> n=0
:: skip -> n=1
:: skip -> n=2
:: skip -> n=3
fi
```

skips are **redundant**, because assignments are themselves **always executable**...

non-deterministic branching



Promela: Statements

```
do
:: choice1 -> stat1.1; stat1.2; stat1.3; ...
:: choice2 -> stat2.1; stat2.2; stat2.3; ...
:: ...
:: choicen -> statn.1; statn.2; statn.3; ...
od;
```

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

Promela: Statements

- Example – modelling a **traffic light**

if- and **do**-statements are ordinary **Promela** statements; so they can be nested.

```
mtype = { RED, AMBER, GREEN } ;
```

mtype (message type) models **enumerations** in Promela

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

Note: this **do**-loop does **not** contain any **non-deterministic** choice.

Promela: Interleaving Semantics

- Promela **processes** execute **concurrently**.
- **Non-deterministic scheduling** of the processes.
- Processes are **interleaved** (statements of different processes do not occur at the same time).
 - exception: **rendez-vous communication**.
- All statements are **atomic**; each statement is executed without interleaving with other processes.
- Each process may have several **different possible actions** enabled at each point of execution.
 - only one choice is made, **non-deterministically**.

Déclaration de canaux

- On l'introduit par le mot clé **chan**, suivi du nom du canal et optionnellement de la longueur du *fifo* et du type des messages qui circulent.
- Par exemple :
 - `chan Ouvreporte=[0] of {byte, bit}, Transfert=[2] of {bit, short, chan};`
- ouvreporte est un canal *synchrone*, car sa longueur est 0, ce qui correspond à un rendez-vous ;
 - sur ce canal circulent des messages ayant une partie byte et une partie bit.
- Transfert est un canal *asynchrone*, car il peut stocker (au plus) deux messages.

Opérations sur les canaux

- Sur un canal on peut
 - envoyer (opération ``!")
 - ou recevoir (opération ``?") des messages.
- Par exemple : `ouvreporte!i,0; ouvreporte?i,1; ouvreporte?eval(etage),1`
 - La fonction ``eval" force l'égalité des valeurs reçues avec `etage`, la variable `etage` n'est pas changée.

Remarques : initialisation des canaux

- `chan a = [16] of { short }`
 - Le canal peut stocker jusqu'à 16 messages du type short.
- `chan c[3] = [4] of { byte }`
 - Un vecteur de 3 canaux, chacun avec une capacité de 4 trames de message, chaque trame est constituée d'un champ de message de type byte.
- Si le message contient plus qu'un champ, la déclaration sera:
- `chan qname = [16] of { byte, int, chan, byte }`
 - Un seul canal qui peut stocker jusqu'à 16 messages, Chaque message consiste de 4 champs

Instruction ``init''

- L'exécution du système commence par le processus init (c'est le main de Promela).

Par exemple :

- `init { run porte(1); run porte(2); run porte(3); run ascenseur() }`

Expressions

- Un expression peut être utilisée comme une instruction si elle ne fait pas des effet de bord (opérations ``--" et ``++" de C).
- Dans ce cas, elle est exécutable quand sa valeur devient vraie (par le changement des valeurs des variables partagées).
- Par exemple : `(a == b);` est équivalent à :
`while (a != b) skip;`

Vérification avec SPIN

- SPIN génère avec l'option -a un analyseur d'un programme PROMELA sous la forme d'un programme c (pan.c).
- Cet analyseur, une fois compilé et exécuté, explore tous les états accessibles du programme PROMELA.
- Exemple :
\$ spin -a phil
\$ cc -o pan pan.c
\$ a.out
- Si une erreur est détectée lors de l'analyse, une trace est sauvgardée dans le fichier pan.trail.
Cette trace est visualisable avec spin en utilisant l'option -t.
- Exemple :
\$ spin -t -p -g phil

Instruction ``atomic''

- L'exécution d'une séquence d'instructions préfixée par ``atomic'' peut être rendue indivisible, c'est-à-dire sans l'entrelacement des actions des autres processus.
- Par exemple : `atomic { run porte(1); run porte(2);
run porte(3); run ascenseur() }`

Xspin

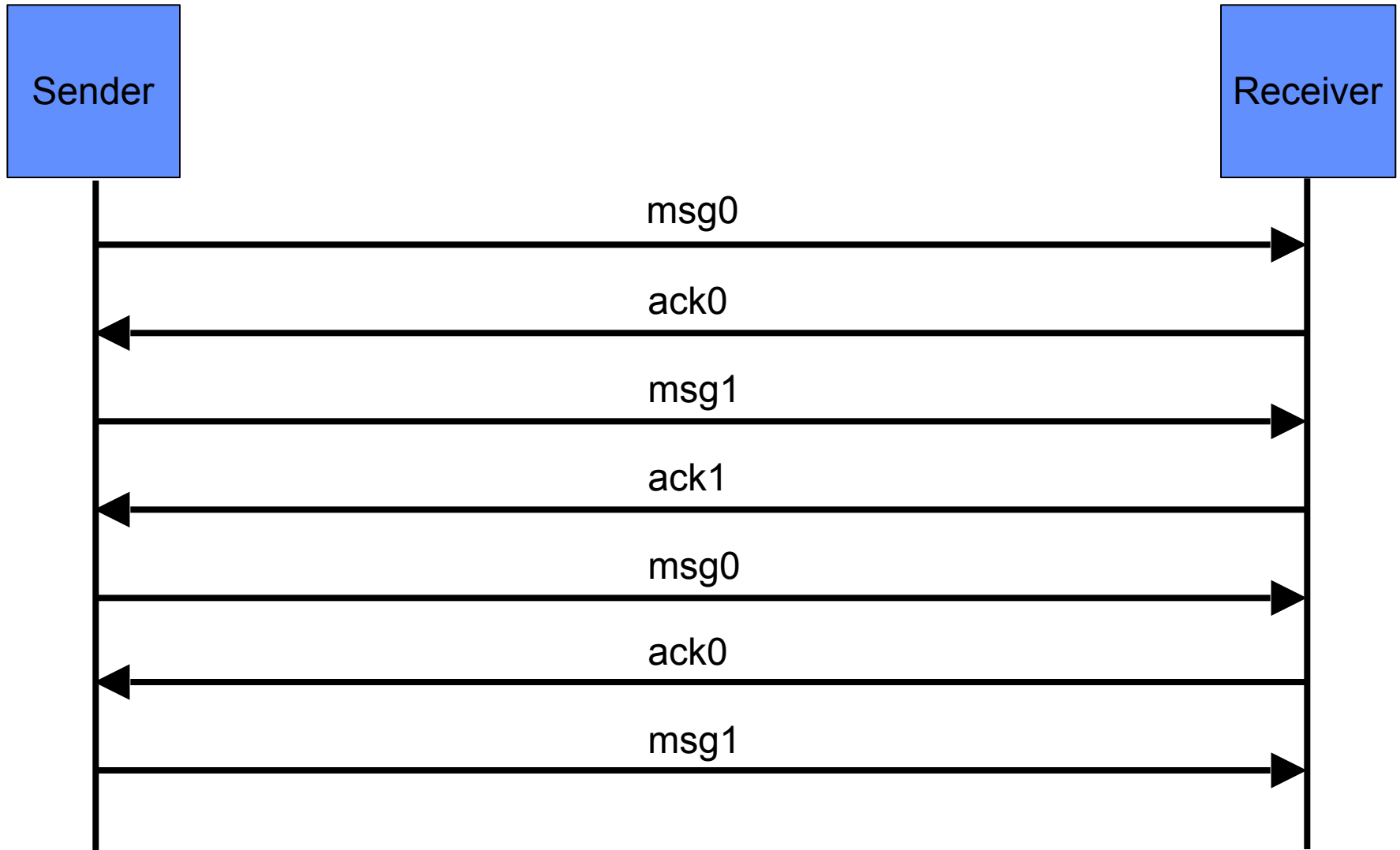
GUI for Spin



Alternating Bit Protocol

- **Two processes want to communicate**
- **They want acknowledgement of received messages**
- **Sending window of one message**
- **Each message is identified by one bit**
- **Alternating values of the identifier**

Alternating Bit Protocol



Alternating Bit Protocol



Alternating Bit Protocol



Sender Process

```
active proctype Sender()  
{  
    do  
        ::  
            if  
                :: receiver!msg0;  
                :: skip  
            fi;  
        do  
            :: sender?ack0 -> break  
            :: sender?ack1  
            :: timeout ->  
                if  
                    :: receiver!msg0;  
                    :: skip  
                fi;  
        od;  
    od;  
}  
::  
    if  
        :: receiver!msg1;  
        :: skip  
    fi;  
    do  
        :: sender?ack1 -> break  
        :: sender?ack0  
        :: timeout ->  
            if  
                :: receiver!msg1;  
                :: skip  
            fi;  
    od;  
}
```

Modèle d'un canal non idéal

- Pour modéliser la possibilité de perte de message, ajouter l'option *skip*
 - *skip* est l'opération nulle de PROMELA.
 - Toujours exécutable et sans effet.

Simulation : SPIN

- Utilisation par défaut SPIN : exécution aléatoire de la simulation du modèle de validation
⇒ pas de vérification de la justesse
- Les exceptions sont les erreurs non tolérées:
 - blocage, réception non spécifiée, violation des assertions.
 - Dans ce cas l'exécution de la simulation s'estompe si une violation d'assertion apparaisse

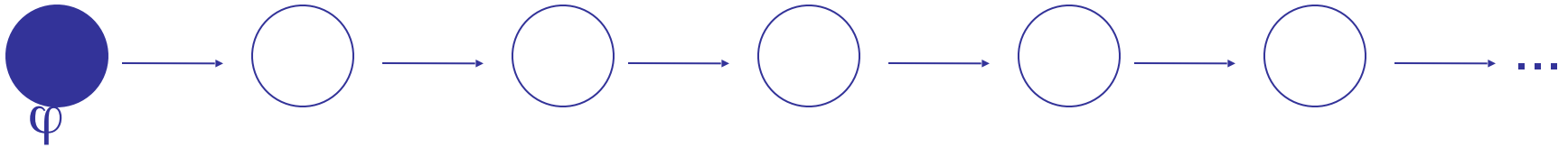
LTL (Linear Temporal Logic)

-safety & liveness

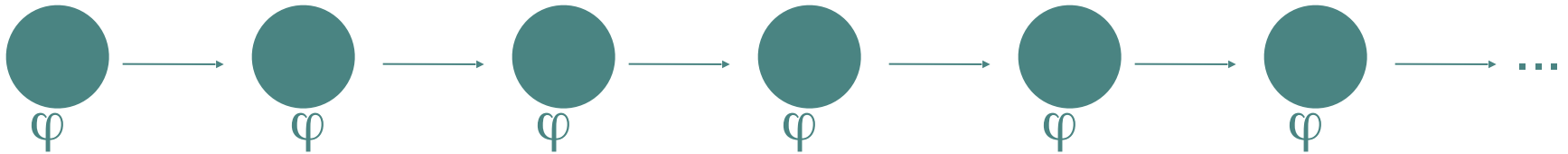
[Pnueli 1977; Lichtenstein & Pnueli 1982]

Properties of Linear Time Temporal Logic

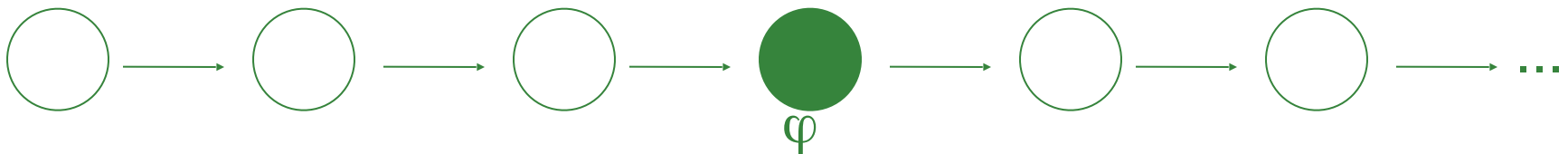
φ holds for a computation iff it holds in the **first** state



$\Box\varphi$ holds for a computation iff φ holds in **all** states

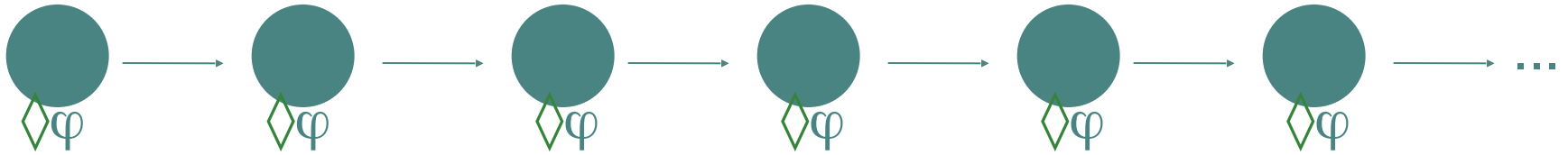


$\Diamond\varphi$ holds for a computation iff φ holds in **some** state

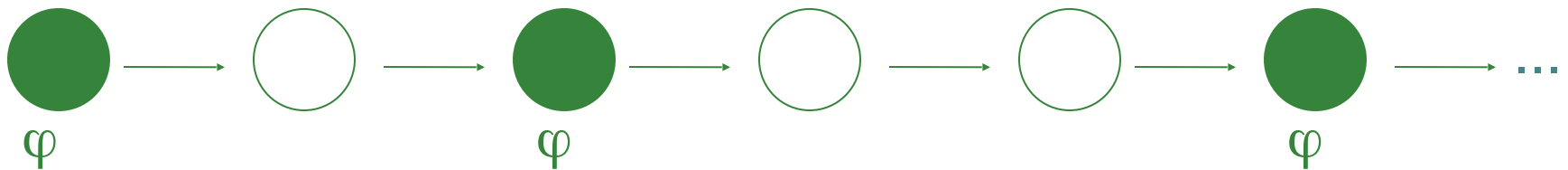


Properties of Linear Time Temporal Logic

$\Box \Diamond \varphi$ holds for a computation iff $\Diamond \varphi$ holds in **all** states



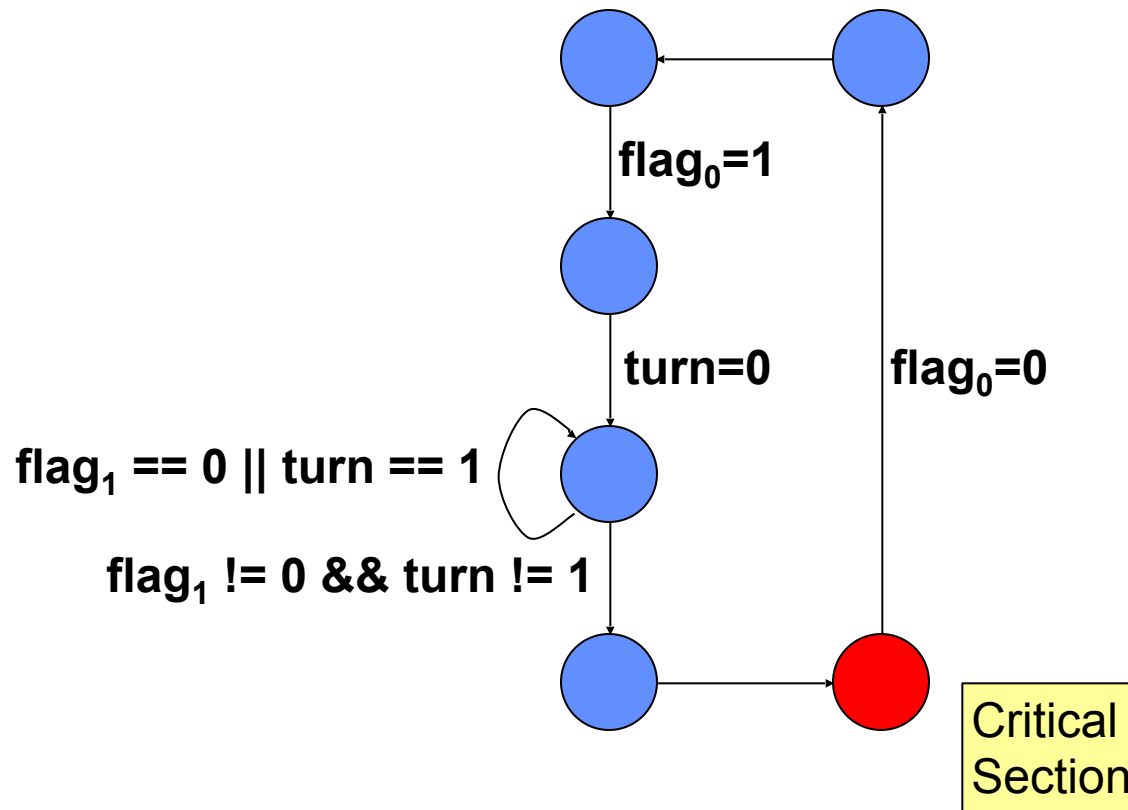
$\Diamond \varphi$ holds in a state iff φ holds in some „later“ state



think for a moment about $\Diamond \Box \varphi$

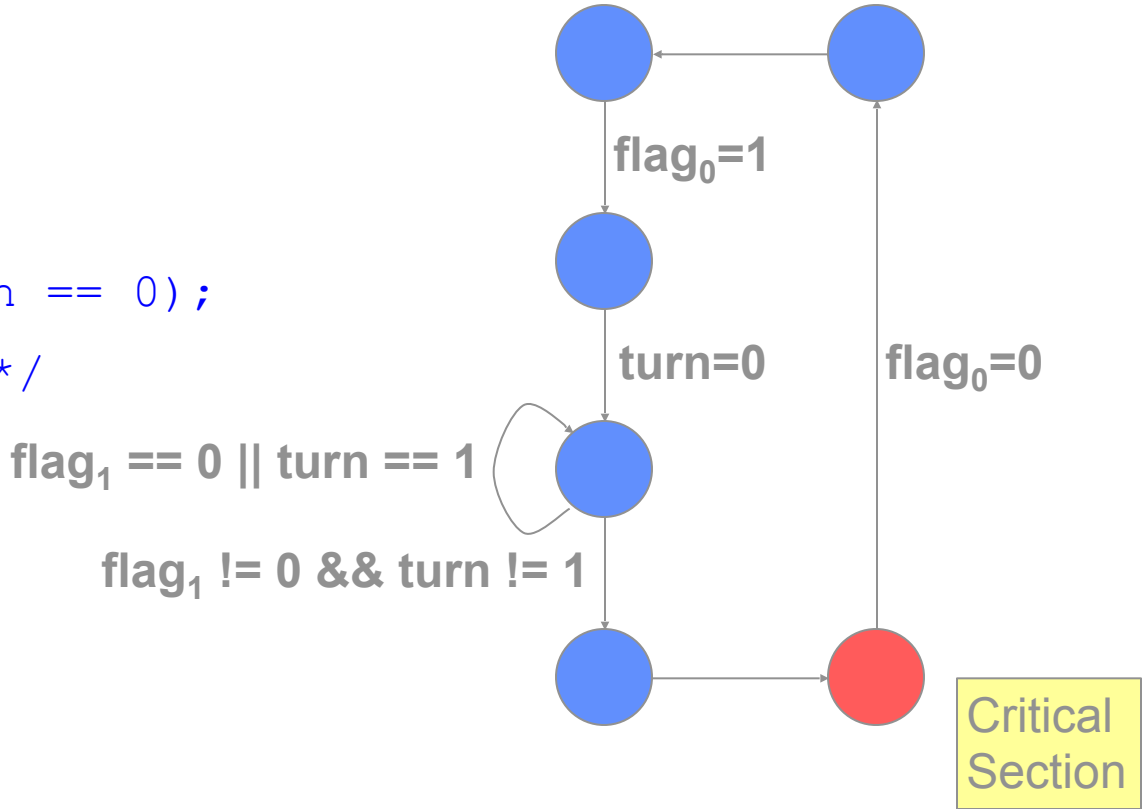
Mutual Exclusion

- Peterson's solution to the mutual exclusion problem



Mutual Exclusion in SPIN

```
bool turn;  
bool flag[2];  
proctype mutex0() {  
again:  
    flag[0] = 1;  
    turn = 0;  
    (flag[1] == 0 || turn == 0);  
    /* critical section */  
    flag[0] = 0;  
    goto again;  
}
```



Mutual Exclusion in SPIN

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

`_pid:`
Identifier of the process

```
active [2] proctype user()
```

```
{  
    assert(_pid == 0 || __pid == 1);
```

`assert:`
Checks that there are only
at most two instances with
identifiers 0 and 1

```
again:
```

```
    flag[_pid] = 1;
```

```
    turn = _pid;
```

```
    (flag[1 - _pid] == 0 || turn == 1 - _pid);
```

```
        /* critical section */
```

```
    flag[_pid] = 0;
```

```
    goto again;
```

```
}
```

Mutual Exclusion in SPIN

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

```
byte ncrit;
```

ncrit:

Counts the number of processes in the critical section

```
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--;
```

assert:

Checks that there is always at most one process in the critical section

```
    flag[_pid] = 0;
```

```
    goto again;
```

```
}
```

- **Generate, compile and run the verifier**
 - to check for deadlocks and other major problems:

```
$ spin -a mutex
$ cc -O pan pan.c
$ pan
full statespace search for:
assertion violations and invalid endstates
vector 20 bytes, depth reached 19, errors: 0
79 states, stored
0 states, linked
38 states, matched total: 117
hash conflicts: 4 (resolved)
(size s^18 states, stack frames: 3/0)
unreached code _init (proc 0);
    reached all 3 states
unreached code P (proc 1):
    reached all 12 states
```

Mutual Exclusion

- **Verifier: Assertion can be violated**
 - Can use -t -p to find out the trace
 - Or use XSpin
- **Another way of catching the error**
 - Have another monitor process ran in parallel
 - Allows all possible relative timings of the processes
 - Elegant way to check validity of system invariant

Mutual Exclusion in SPIN

```
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++;
    /* critical section */
    ncrit--;

    flag[_pid] = 0;
    goto again;
}

active proctype monitor()
{
    assert (ncrit == 0 || ncrit == 1) }
```

Finally,

- **Can specify an LTL formula and run the model-checker**

Example:

```
#define p count <= 1
```

```
– LTL claim: [] p
```

- **Note: all variables in LTL claims have to be global!**
- **LTL claim gets translated into NEVER claim and stored either in .ltl file or at the end of model file**
 - Only one LTL property can be verified at a time
- **Parameters can be set using XSpin**
 - Depth of search, available memory, etc.

Mutual Exclusion in SPIN

```
bool turn, flag[2];
bool critical[2];

active [2] proctype user()
{
    assert(_pid == 0 || __pid == 1);
again:
    flag[_pid] = 1;
    turn = _pid;
    (flag[1 - _pid] == 0 ||
     turn == 1 - _pid);

    critical[_pid] = 1;
    /* critical section */
    critical[_pid] = 0;

    flag[_pid] = 0;
    goto again;
}
```

LTL Properties:

[] (critical[0] || critical[1])

[] <> (critical[0])

[] <> (critical[1])

[] (critical[0] ->
 (critical[0] U
 (!critical[0] &&
 (!!critical[0] &&
 !critical[1]) U critical[1])))

[] (critical[1] ->
 (critical[1] U
 (!critical[1] &&
 (!!critical[1] &&
 !critical[0]) U critical[0])))

Note: critical[] is a global var!

Alternatively,

```
#define p ncrit <= 1
#define q ncrit = 0
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++;
    /* critical section */
    ncrit--;

    flag[_pid] = 0;
    goto again;
}
```

LTL Properties:

$\Box (p)$
 $\Box \langle \rangle (!q)$

Command Line Tools

- **Spin**

- Generates the Promela code for the LTL formula

```
$ spin -f "[ ]<>p"
```

- The proposition in the formula must correspond to **#defines**

- Generates the C source code

```
$ spin -a source.pro
```

- The property must be included in the source

- **Pan**

- Performs the verification

- Has many compile time options to enable different features

- Optimized for performance