Model Verification and Validation

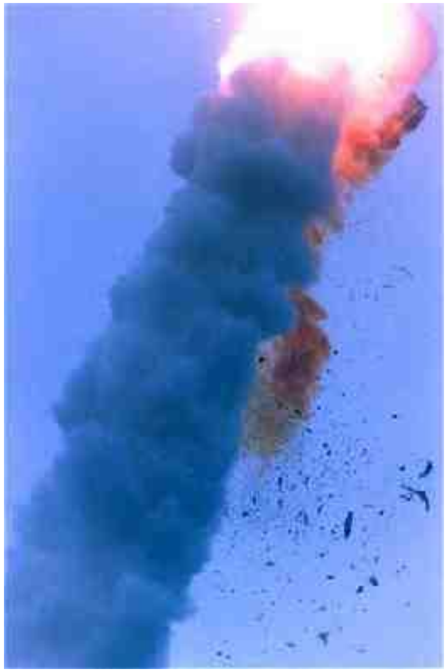
Budapest University of Technology and Economics
Fault Tolerant Systems Research Group

Ariane 5 Booster

- The strongest European booster

Ariane 5 Booster

- On 4 June 1996 it destroyed itself 37 seconds after launch
  - Four satellites were destroyed
  - Loss of $370 million



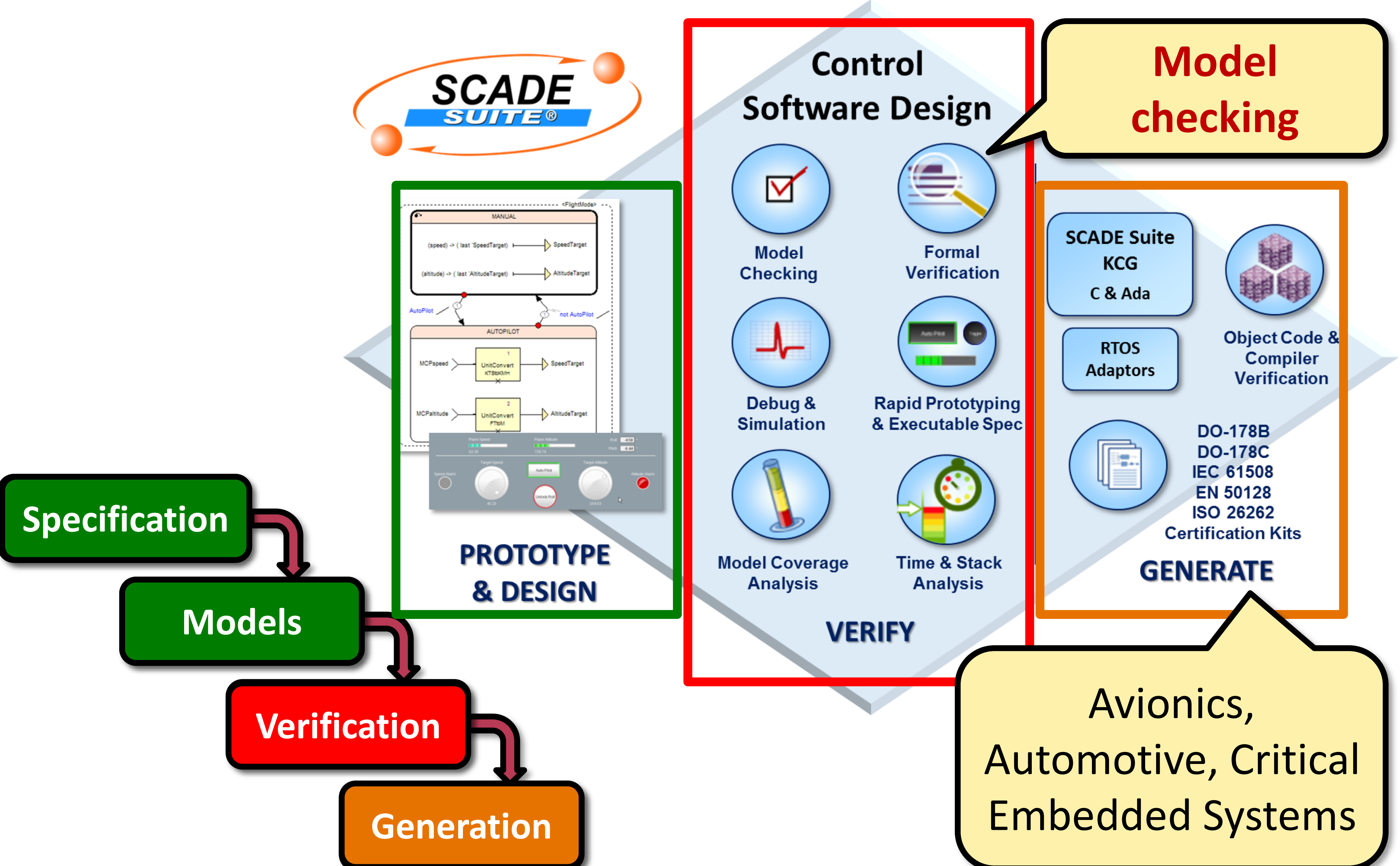
Ariane 5 Booster

- On 4 June 1996 it destroyed itself 37 seconds after launch
  - Four satellites were destroyed
  - Loss of $370 million
- (One of the) world's most expensive software fault
  - Immediate reason:

    Unsuccessful conversion between 64 bit and 16 bit number
  - Underlying reason:

    Modules were never tested together

Example: Esterel SCADE



Basic Concepts | Static Analysis | Testing | Formal Verification

CONTENT

Motivation: Model Life Cycle

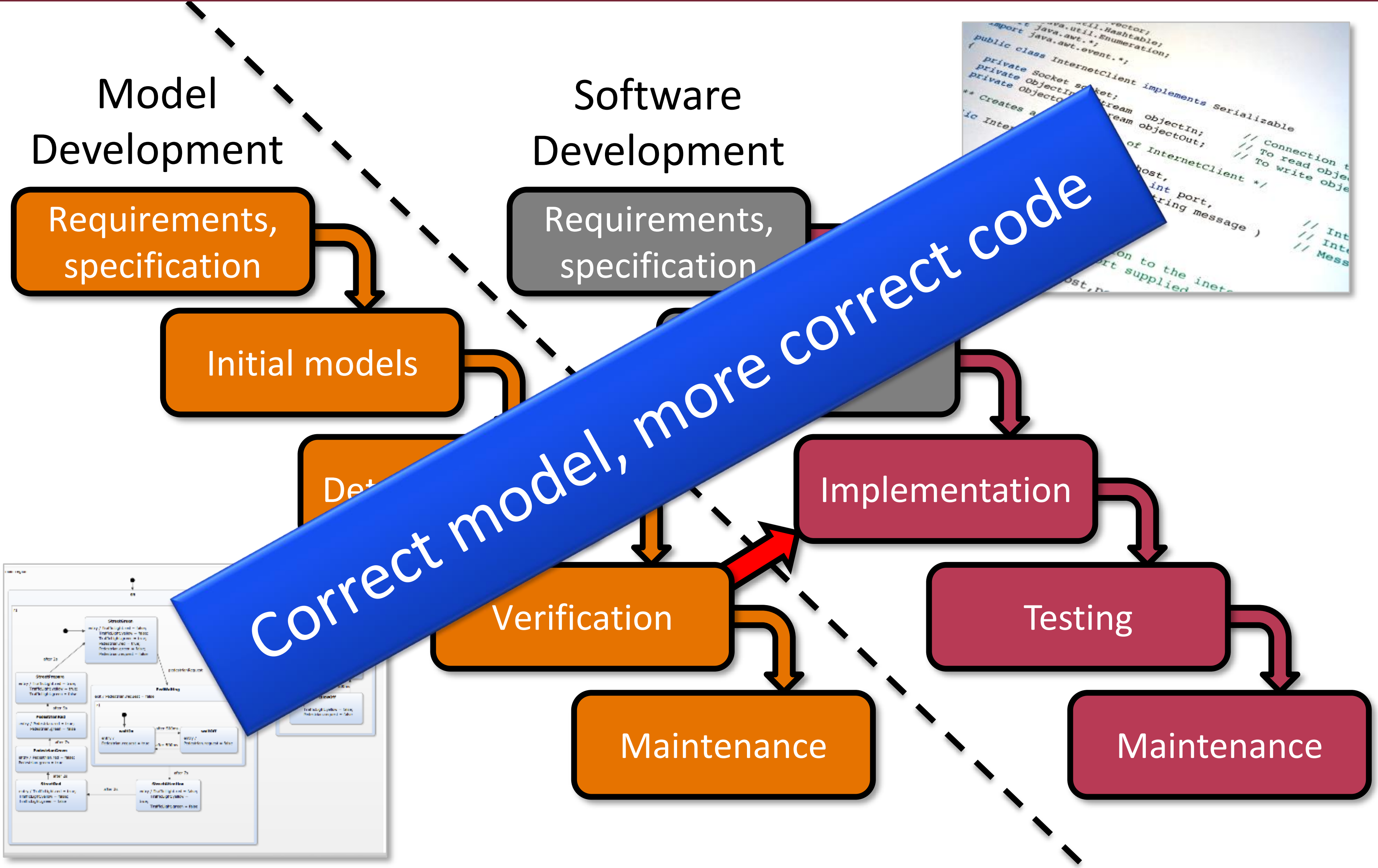
Model Development

- Requirements, specification
- Initial models
- Detailed models
- Verification
- Maintenance

Software Development

- Requirements, specification
- Design
- Implementation
- Testing
- Maintenance

Automatic Code Generation



BASIC CONCEPTS

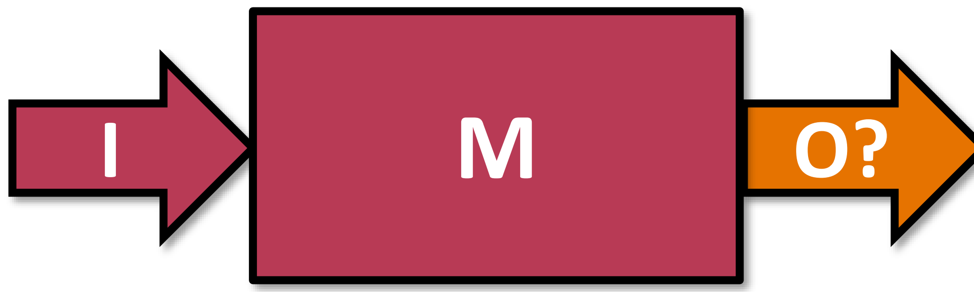
Models and Activities

- Synthesis:

  Model conformant to specification?

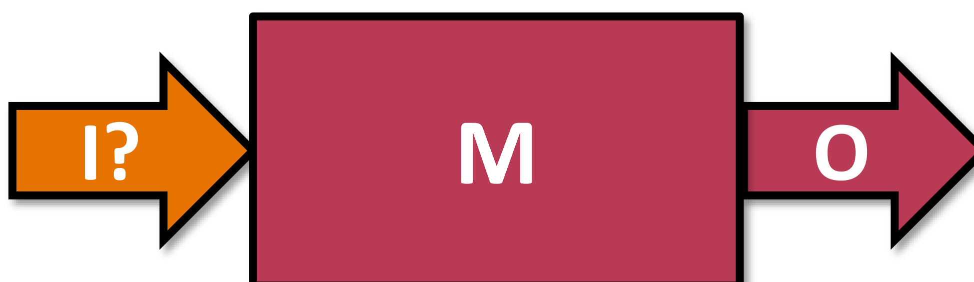
- Analysis:

  Model's behaviour?



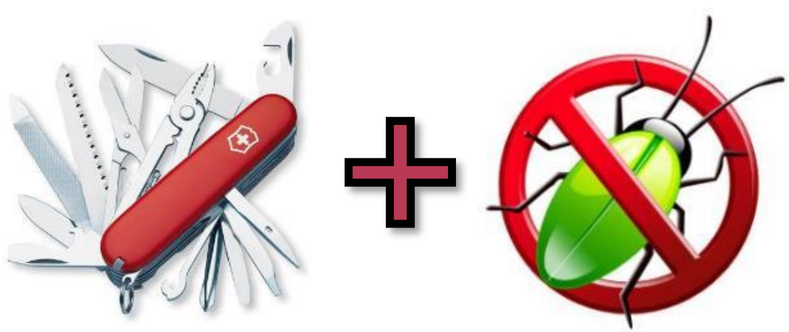
- Control:

  How can the desired state be reached?



Correctness

- **Correctness:**
  model/code fulfils the requirements
  - **Functional Correctness:**
    satisfying the functional requirements
  - Checking non-functional requirements:
    see lecture on Performance modelling
- **Aspects:**
  - Always able to complete the task
  - Error-free
  - No forbidden behaviour



Classification of Functional Requirements

- **Allowed** behaviour (e.g. safety):
  - „Something bad is never true”
  - What state can/can’t be the current state of the sytem
  - What behaviour is prohibited
  - Universal requirements
    - They must always be true
- **Expected** behaviour (e.g. liveness):
  - „Something good eventually happens”
  - What states should be able to be reached
  - What functions should the system be capable of
  - Existential requirements
    - Possibility of fulfilling, potential reachability

Classification of Functional Requirements

- **Allowed** behaviour (e.g. safety):
  - „Something bad is neve… …”
  - What state can/can't be the …
  - What behaviour is prohibited
  - Universal requirements
    - They must always be true
- **Expected** behaviour (e.g. liveness):
  - „Something good eventu… ha…
  - What states should be able t…
  - What functions should the syst…
  - Existential requirements
    - Possibility of fulfilling, potential r…

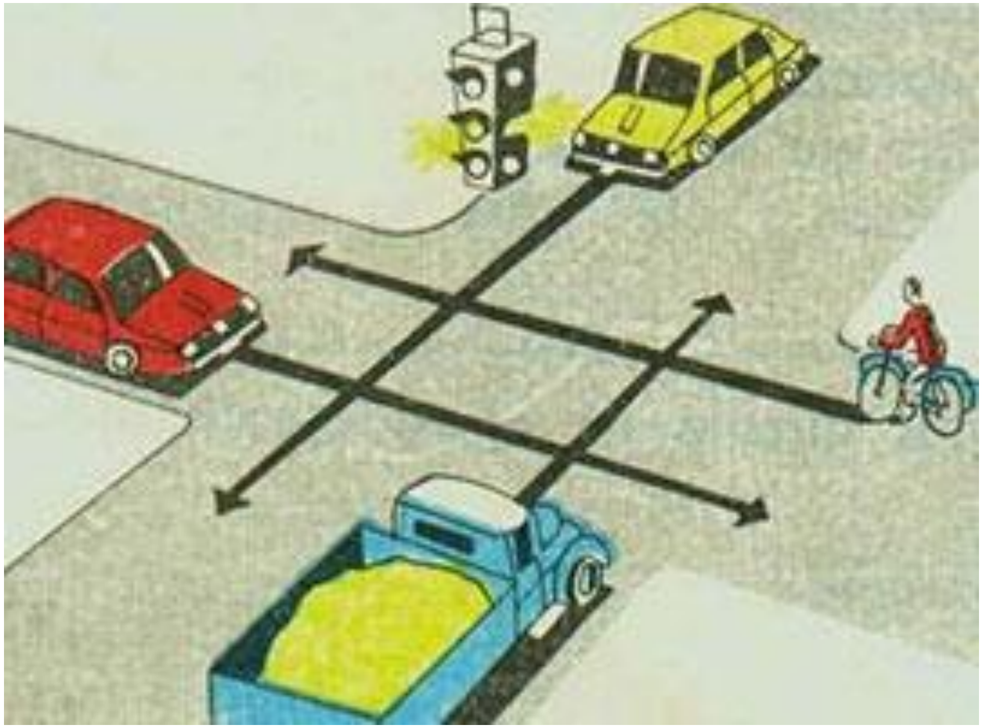
„Traffic lights of crossroads **can never** all **be** green at the same time.”

„The light **should be able to** switch to green.”

Deadlock

Deadlock: A subset of the state space, which cannot be left by the system without external assistance.

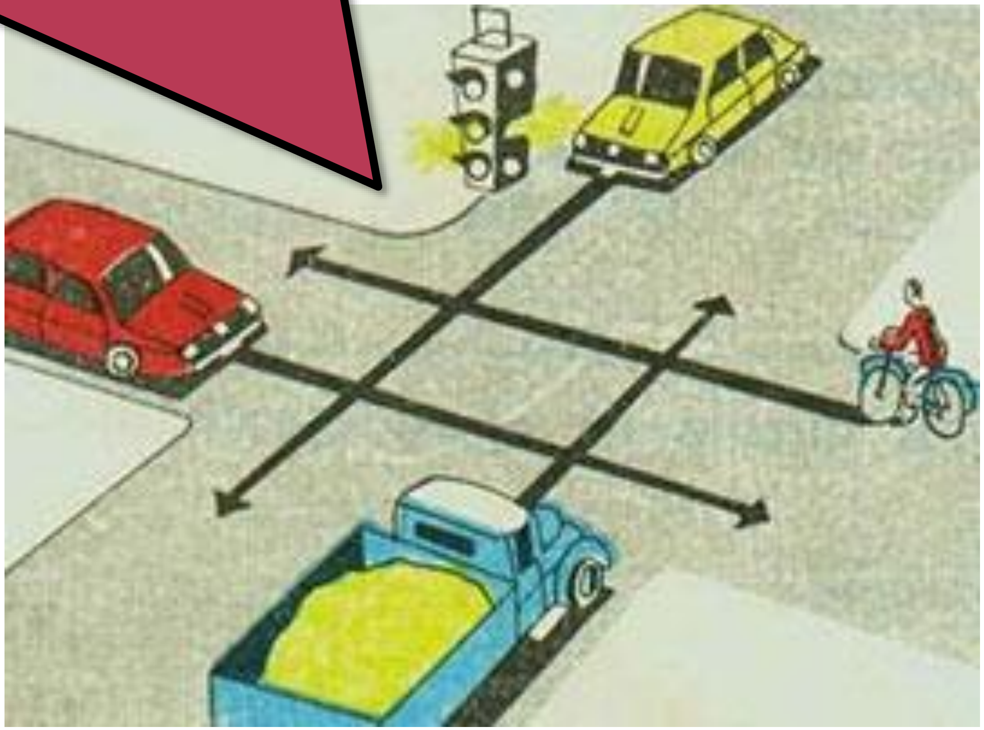
- e.g. Processes waiting for each other



Deadlock

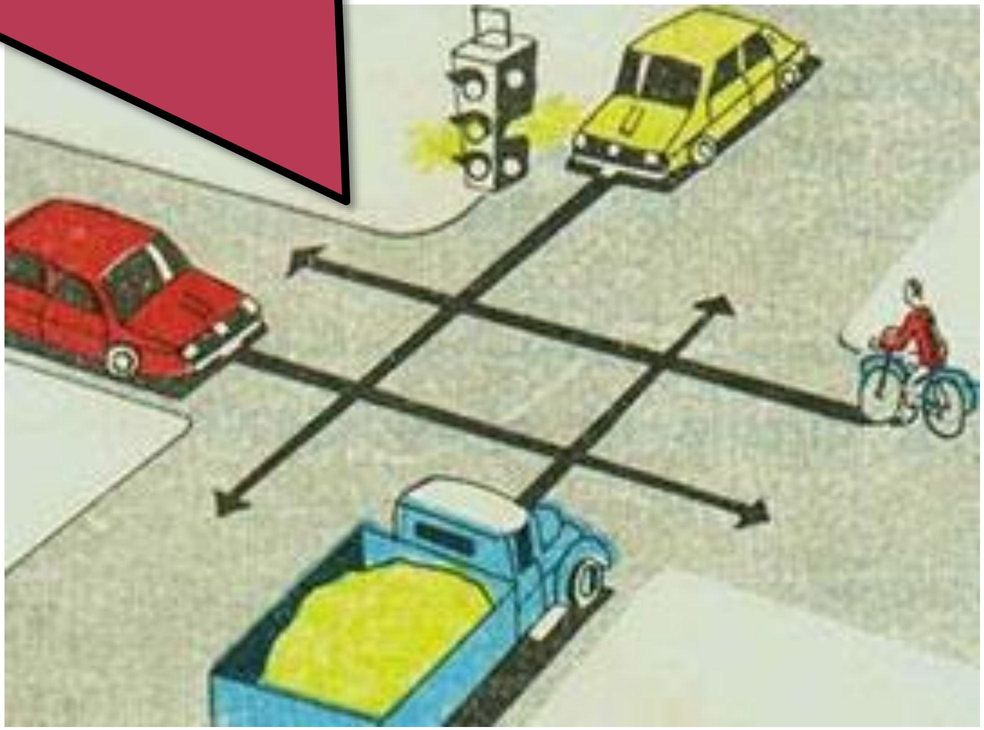
At crossroads – unless road signs or traffic rules tell otherwise – the vehicle coming from the right has right of way [priority].

(Road Traffic Act I, 1988)



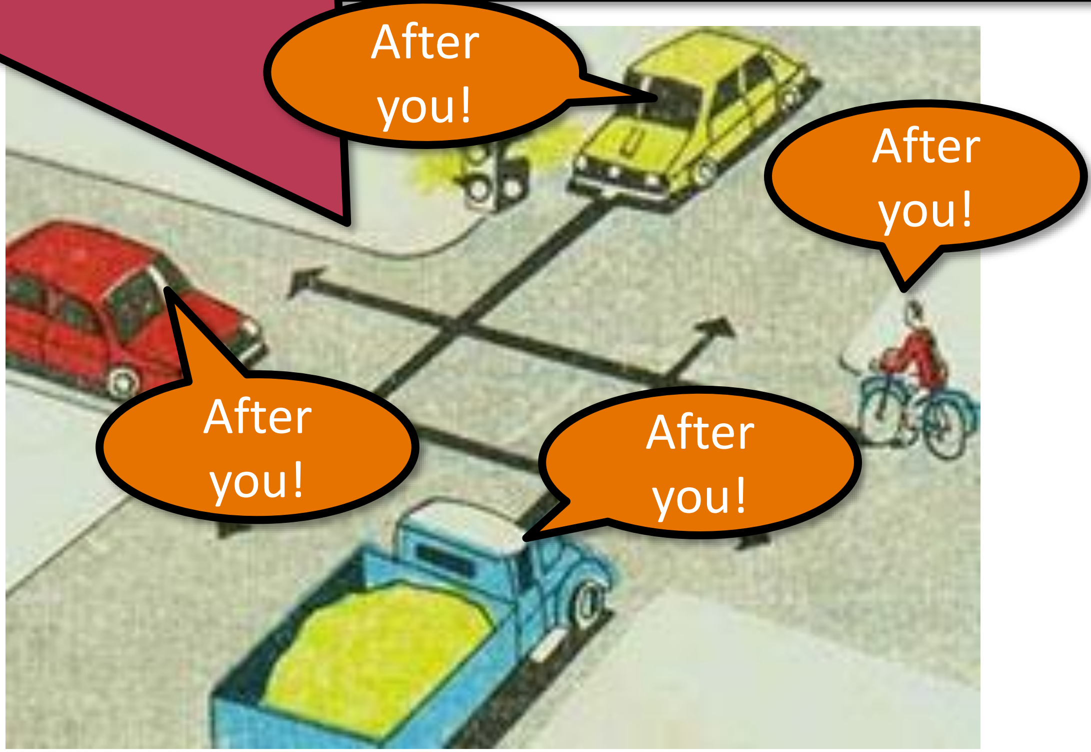
Unlocking the Deadlock

If 4 cars arrive to the crossroad at the same time, then one of them has to disclaim his priority, and let the others go. Otherwise they will stay there forever according to Highway code.



Unlocking the Deadlock

If 4 cars arrive to the crossroad at the same time, then one of them has to disclaim his priority, and let the others go. Otherwise they will stay there forever according to Highway code.



Another Deadlock

If 4 cars arrive to the crossroad at the same time, then one of them has to disclaim his priority, and let the others go. Otherwise they will stay there forever according to Highway code.

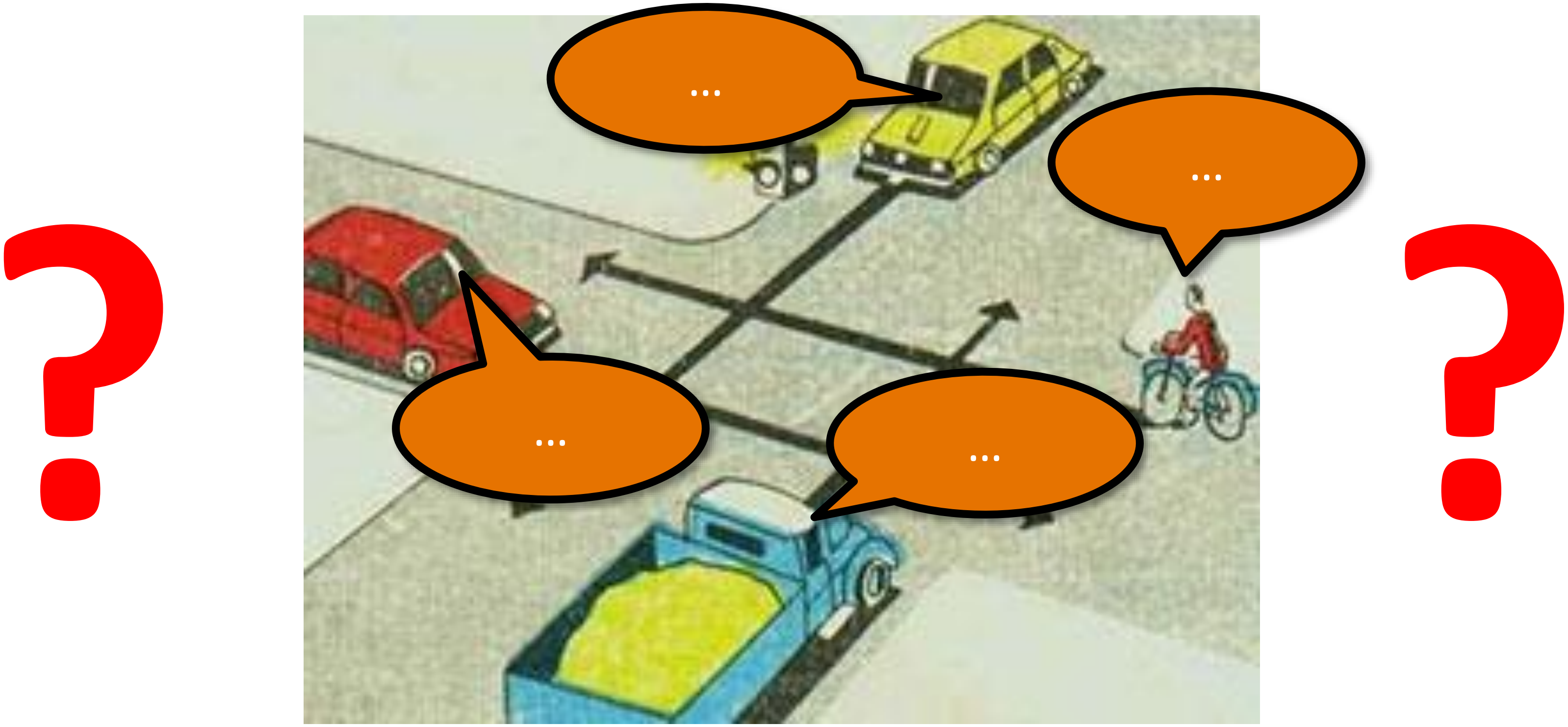
Unlocking the deadlock because of unlocking:

- Asymmetric algorithms
- Algorithms with randomization
  - See the backoff algorithm at Ethernet networks

Infinite Loop (livelock)

Deadlock: Another subset of the state space, which cannot be left by the system without external assistance.

- e.g. result of unlocking the deadlock
- e.g. the Google car with the fixie

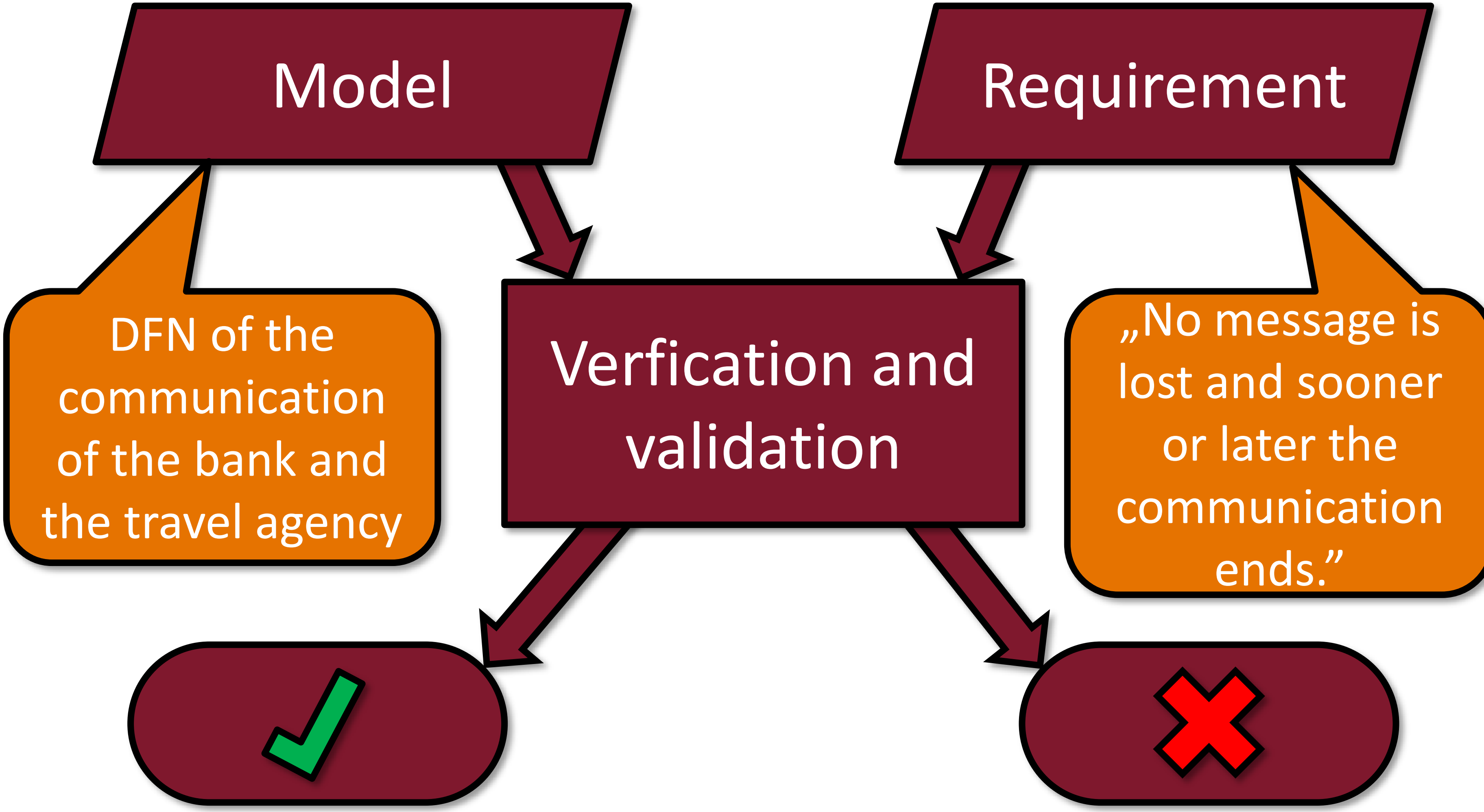


Deadlock

- Common design mistake at parallel systems
  - Often it is difficult to avoid or unlock it
    - The solution believed to be good can also cause problems
  - Difficult to test, may seem random
  - "Multi-core CPU crisis"

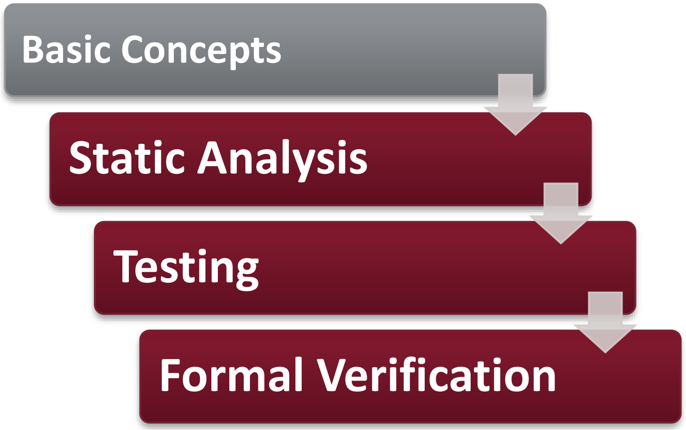
- Examples
  - Two processes have to exchange messages but both are waiting for the other’s message
  - Both of two processes need two of the resources to continue, but each have reserved one

Model Verification and Validation



Types of Analysis

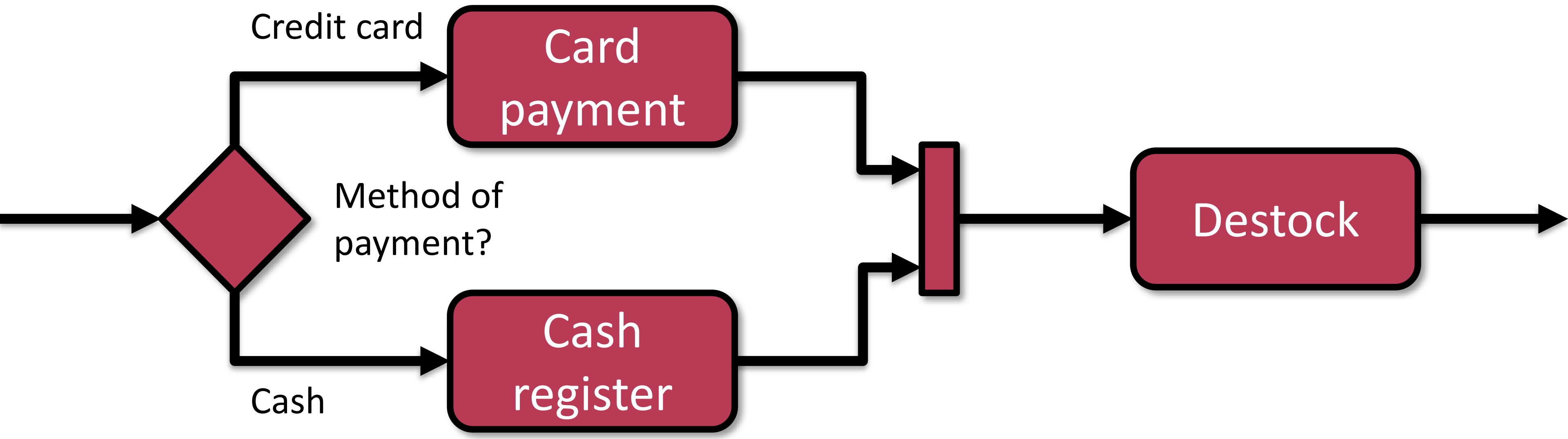
- By goal:
  - **Verification:**
    Am I building the system **the right way**?
    - Is the implementation conformant to the specification?
  - **Validation:**
    Am I building the **right system**?
    - Does the system satisfy the user requirements?
- By method:
  - Static analysis
  - Dynamic analysis
    - „spot check” (testing, simulation)
    - Complete (model checking)



STATIC ANALYSIS

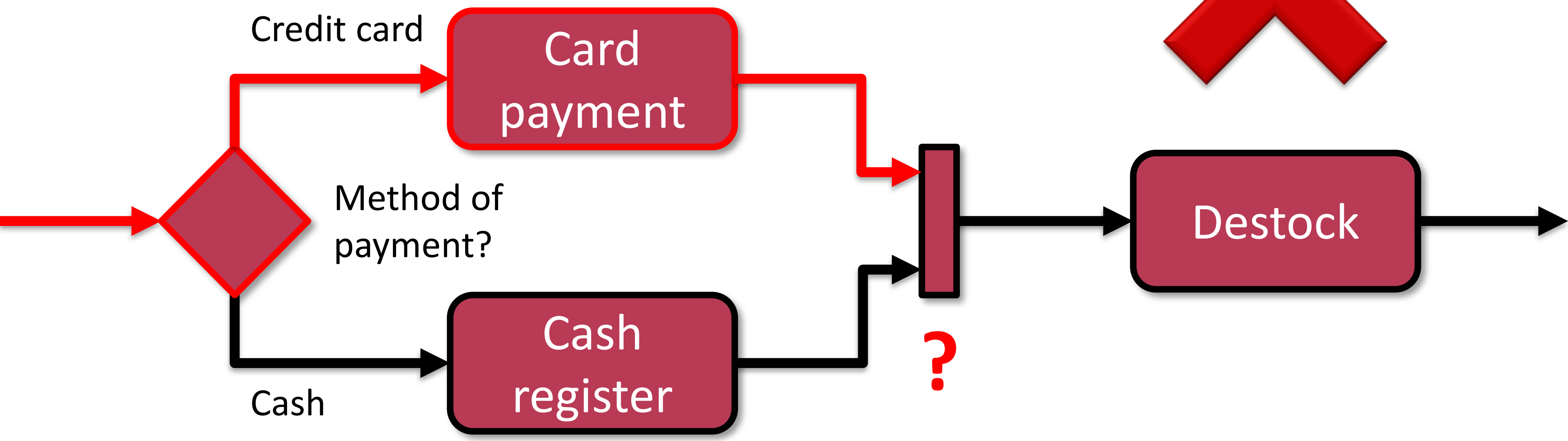
Decision and Join

- Is the following model correct?



Decision and Join

- Is the following model correct?

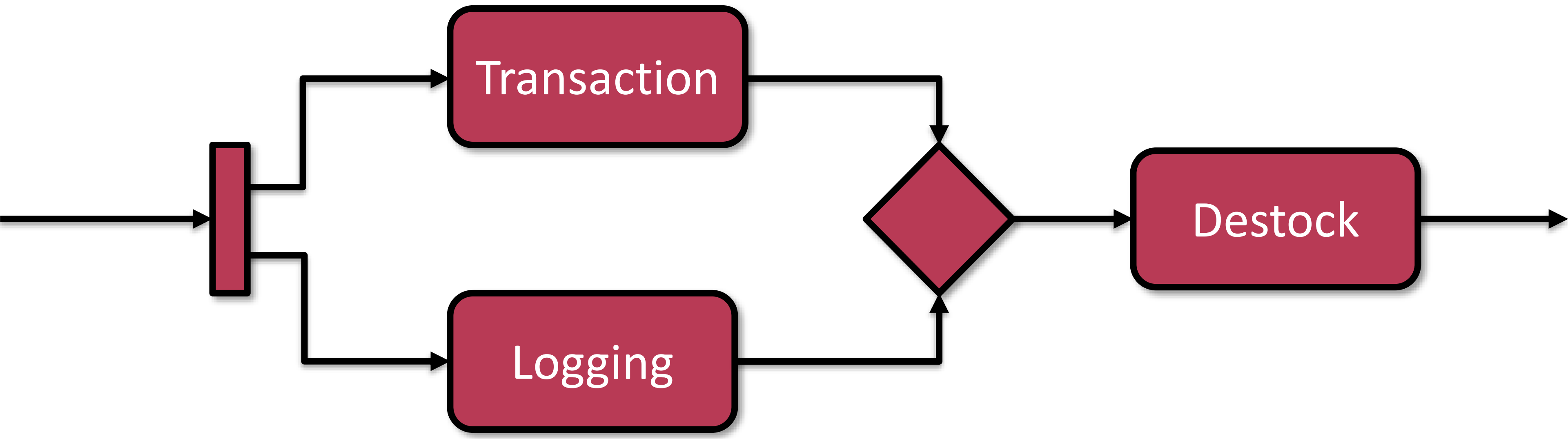


- Join: only continues when tokens arrived from all inputs

→ **DEADLOCK**

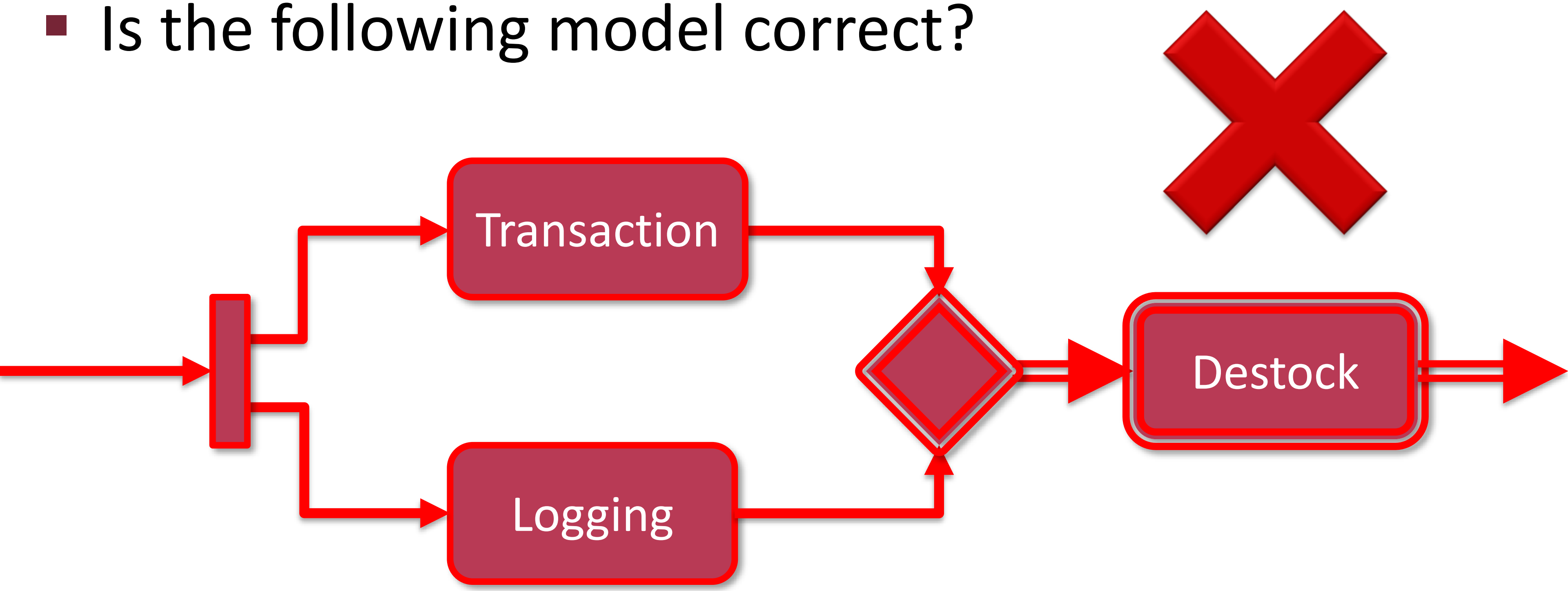
Fork and Merge

- Is the following model correct?



Fork and Merge

- Is the following model correct?

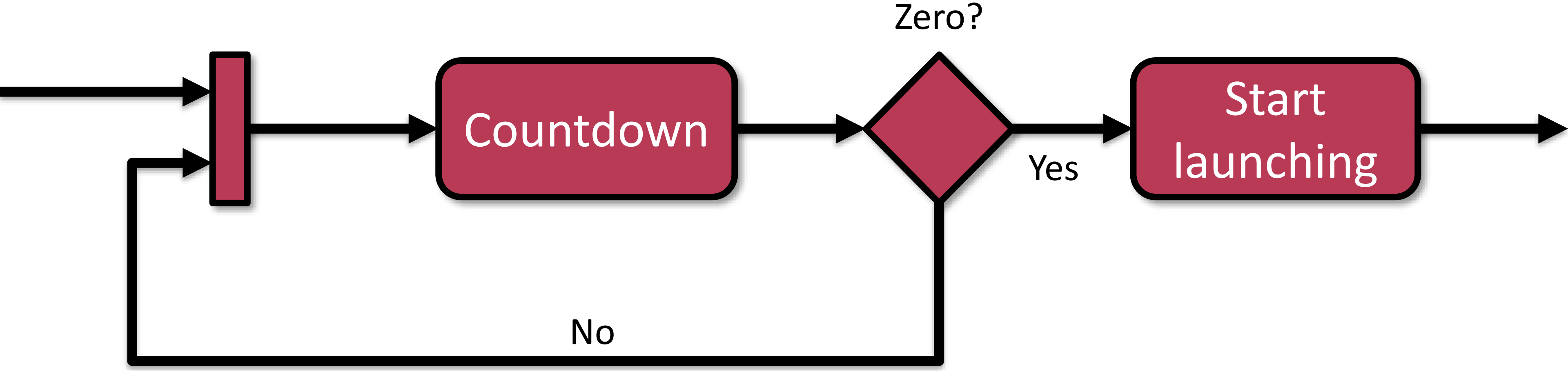


- Merge: let tokens pass through from any branch
  - Doesn't synchronize

→ „Destock" is executed twice

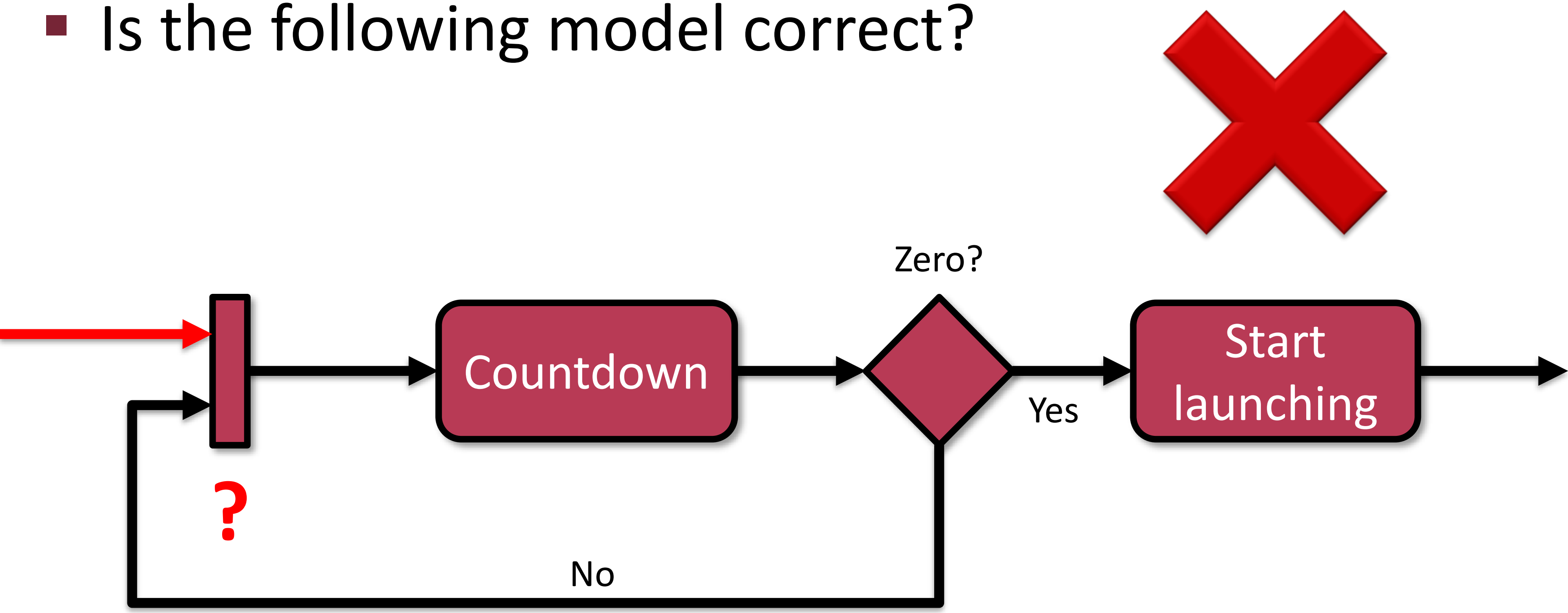
Loop 1.

- Is the following model correct?



Loop 1.

- Is the following model correct?

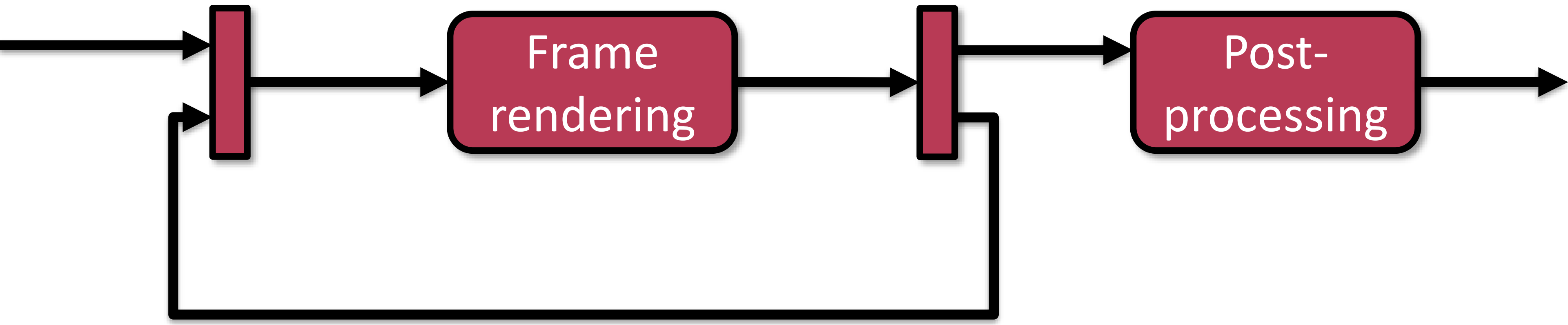


- Join: only continues when tokens arrived from all inputs

→ **DEADLOCK**

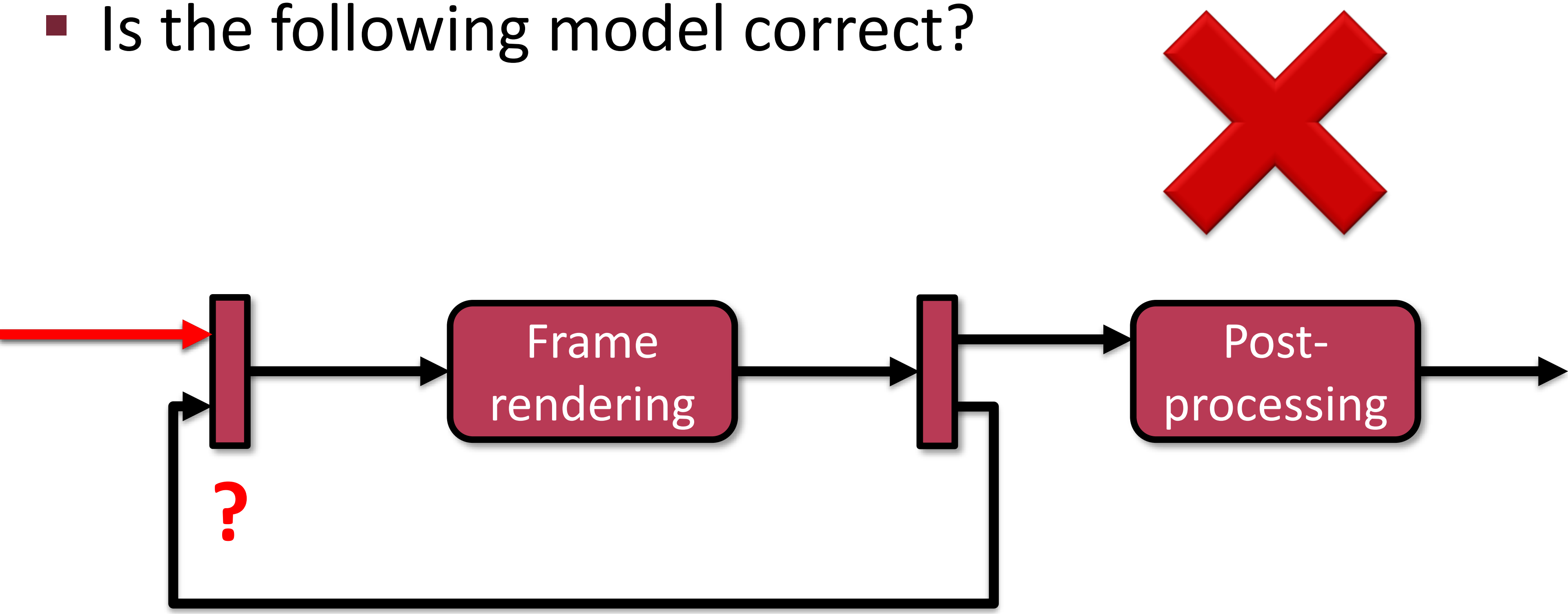
Loop 2.

- Is the following model correct?



Loop 2.

- Is the following model correct?

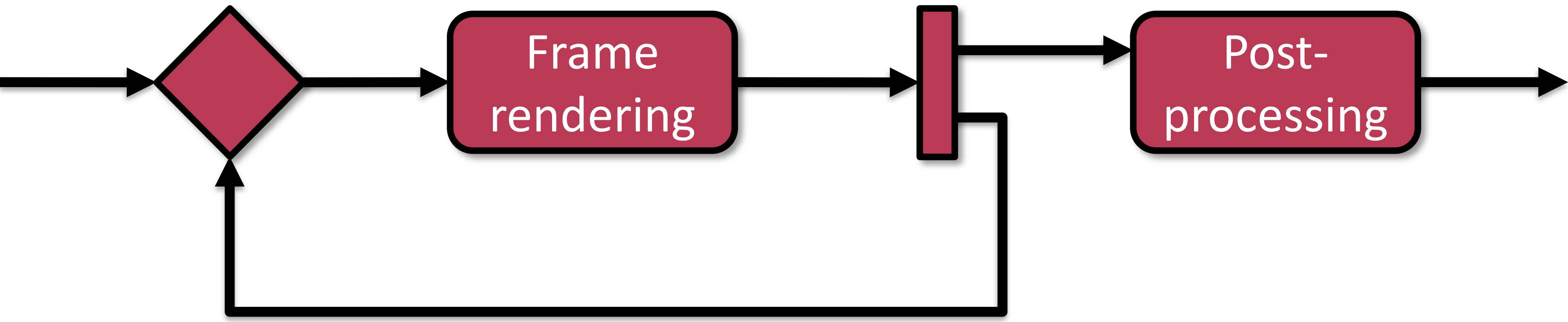


- Join: only continues when tokens arrived from all inputs

→ **DEADLOCK**

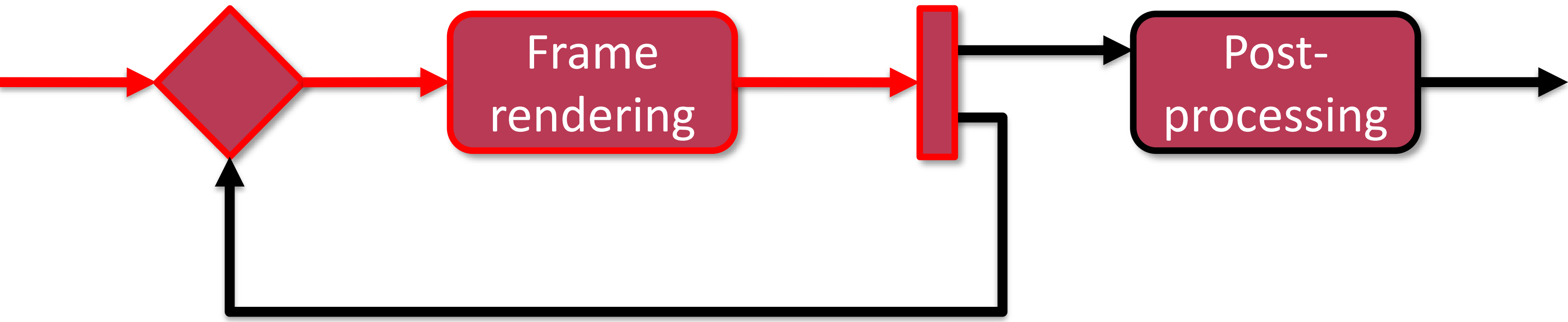
Loop 3.

- Is the following model correct?



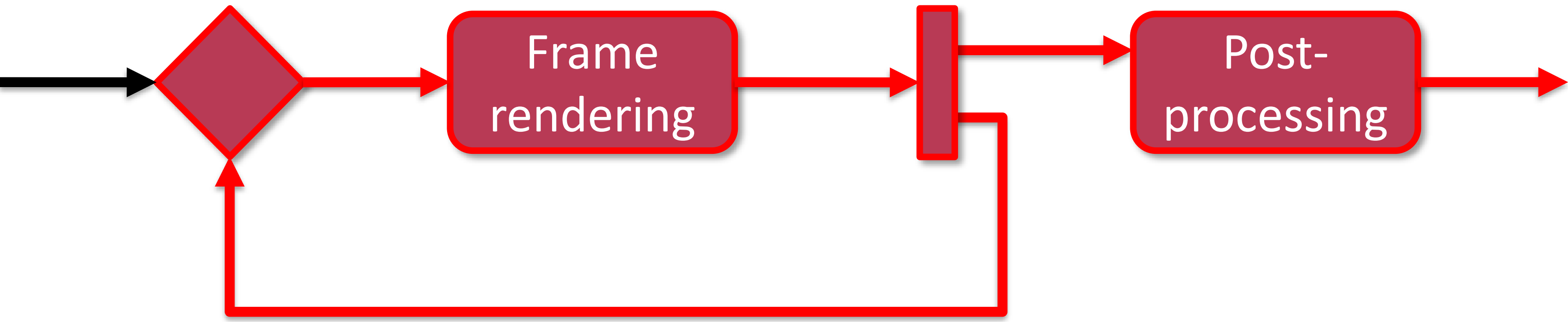
Loop 3.

- Is the following model correct?



Loop 3.

- Is the following model correct?

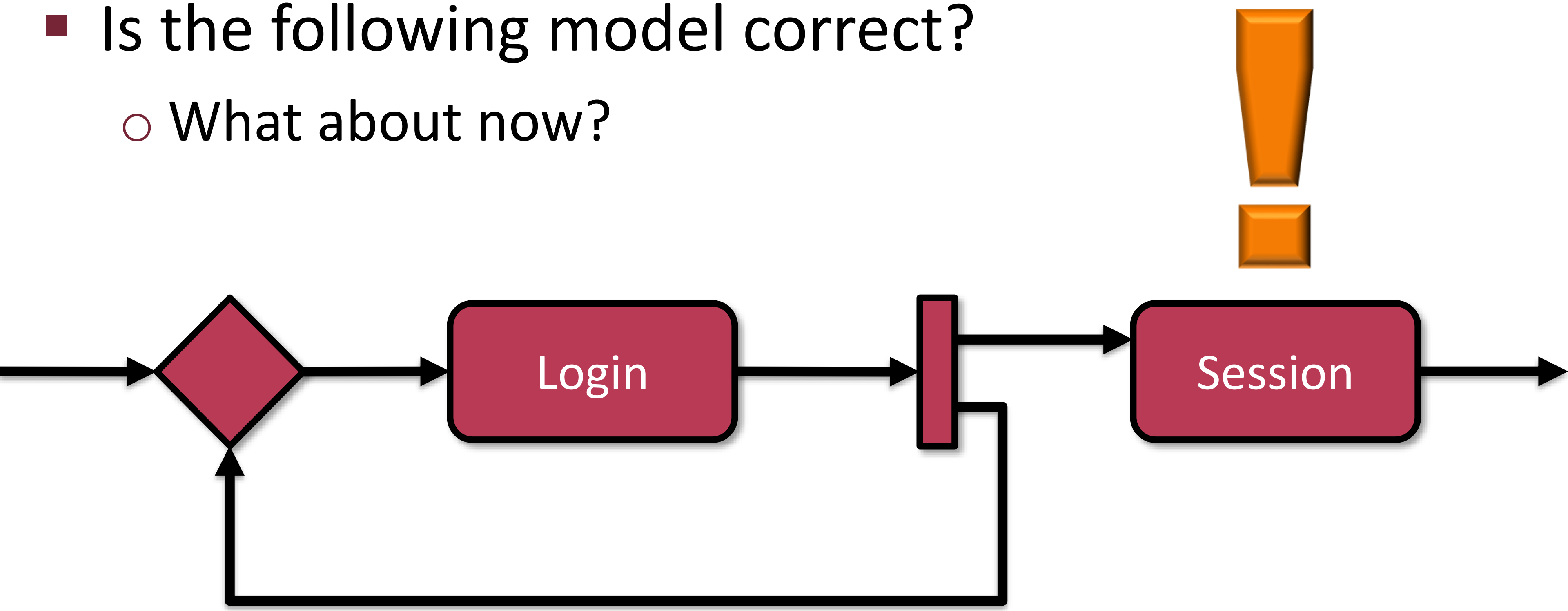


- New frame in every iteration
  - Postprocessing each (many times – how many?)

Borderline case…

Loop 3.

- Is the following model correct?
  - What about now?

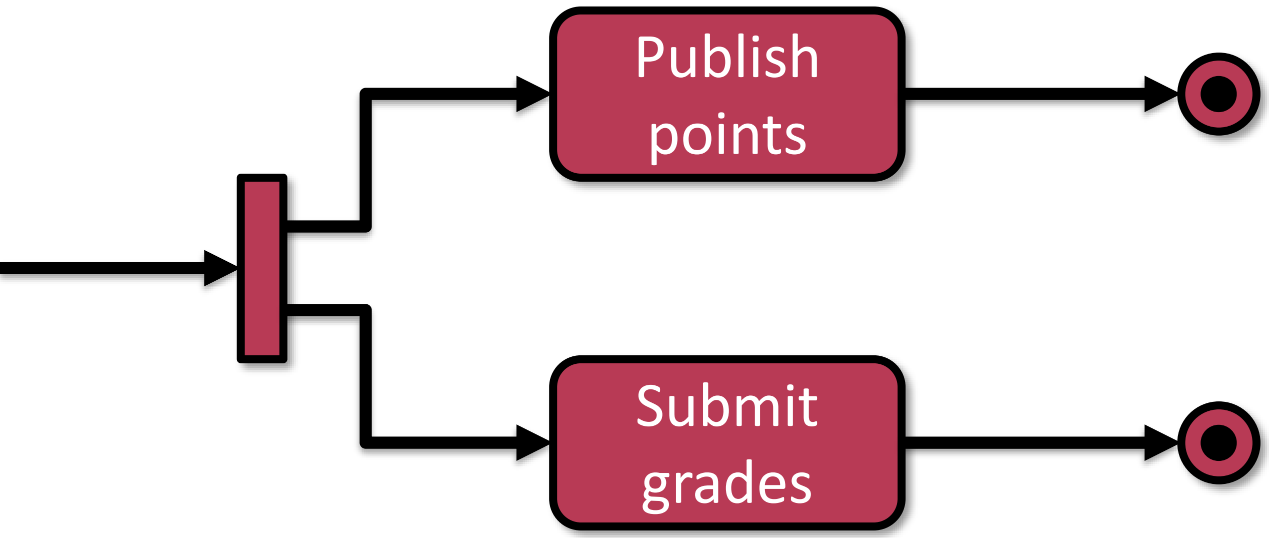


- New login after every login…
  - …and a session…?

→ Faulty implementation „produces" threads

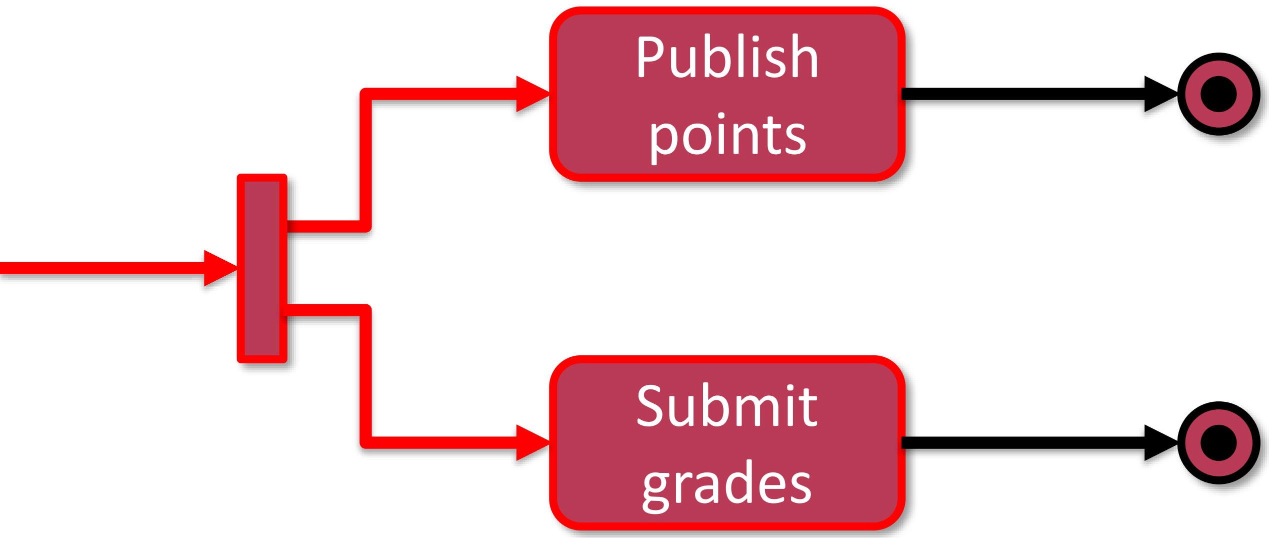
Terminating Node

- Is the following model correct?



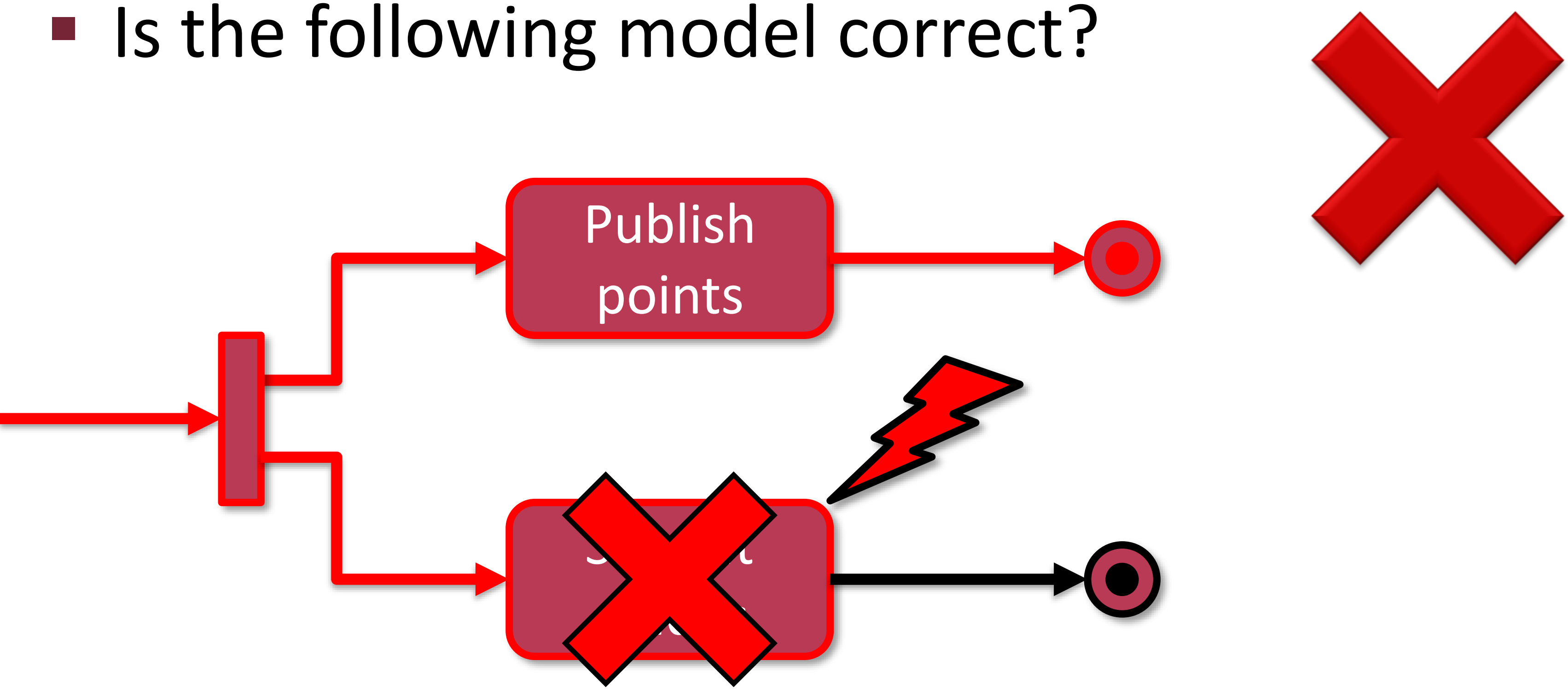
Terminating Node

- Is the following model correct?



Terminating Node

- Is the following model correct?



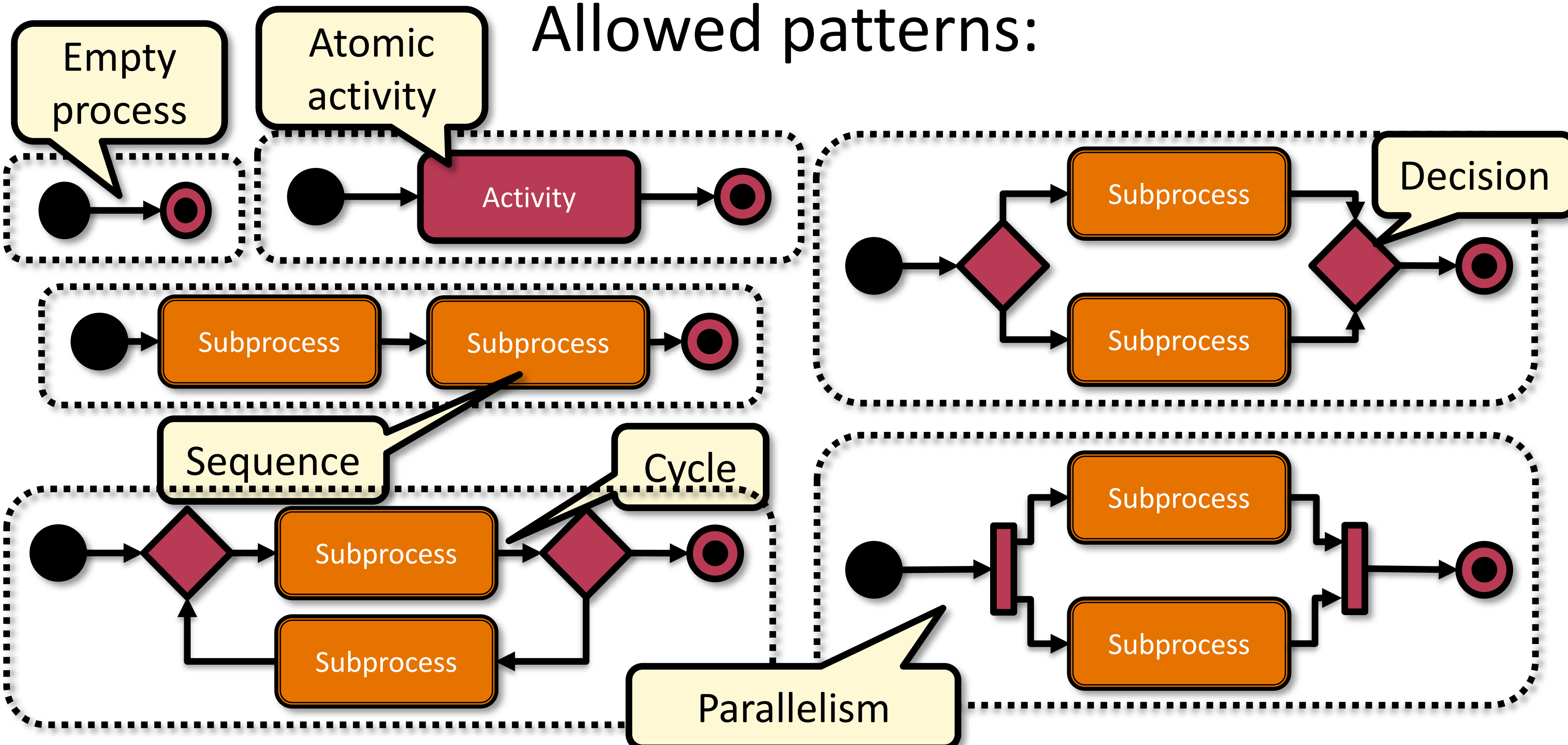
- Terminating node: stops the **complete** process immediately

→ The other activity won't be executed

Well Structured Process Models

- **Lecture:** These problems can be avoided by using **well-structured** processes

Allowed patterns:



Static Analysis of Data Flow

- A process multiplies two numbers
  - Derived requirement:
    - „If at least one of them is even, the result will also be.”
  - Can be traced through the code
    - „Executing in mind”
- **Symbolic execution**
  - Instead of concrete values of variables, the program is executed with sets of possible values
  - Interesting inputs can be defined
    - E.g. Internal branches

      →By what inputs can the branches be reached?

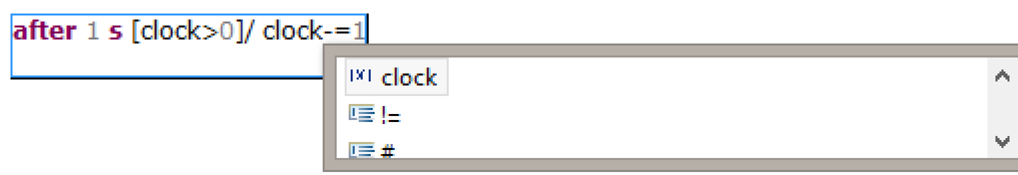
Static Analysis: Syntax Analysis

- Syntax analysis: modelling tools connect logically cascading model elements

Declaration in interface:
var clock: integer = 60

Usage in model:
after 1 s [clock>0]/ clock -= 1

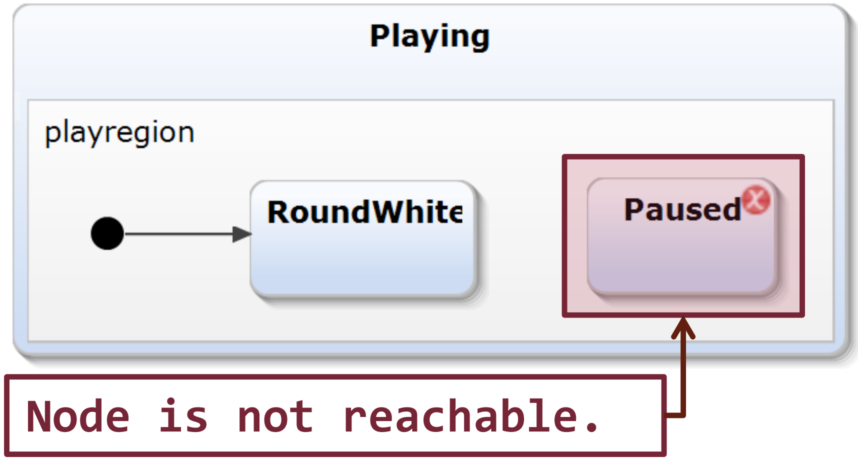
- Syntax-driven editor
  - Fault during editing→ `Couldn’t resolve reference`
  - Advanced editor (offering possibilities for instance)
- Code and diagram together



- Programming: **incorrect** during editing
  Modelling: **correct** during editing

Static Analysis: Structural Correctness

- Structural analysis: examining model graph
- Looking for error-patterns during editing
- Unreachable state, for instance:



- Further analysis: missing initial state, deadlock, variable assignment, etc.

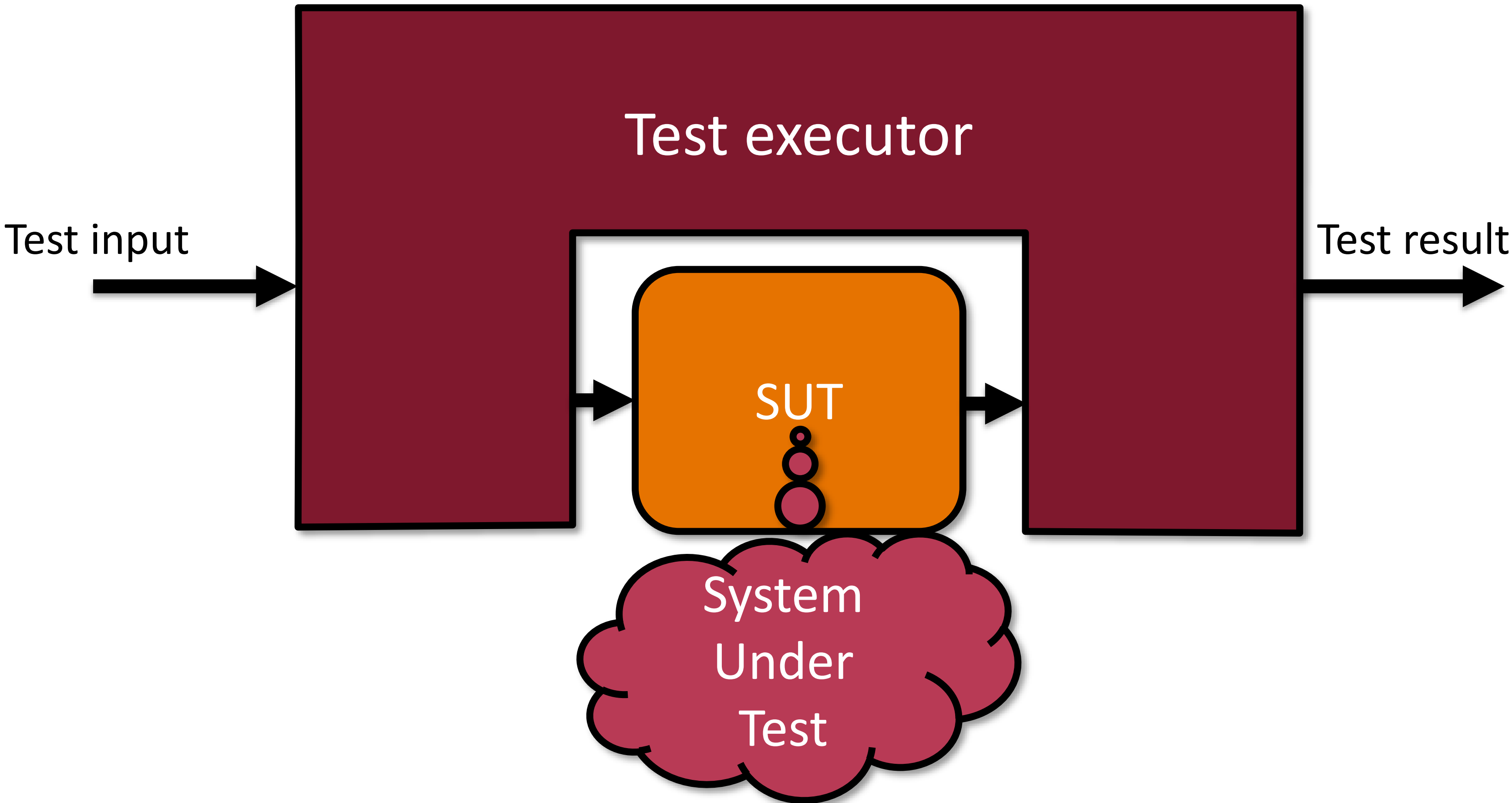
Static Analysis: Supporting Design Rules

- Supporting design guidelines:
  Further rules can be added to the model
  - *Always* and *Oncycle*: Events firing on each clock tick
  - Arbitrary frequence → Typically a malfunction

Using *Always* and *Oncycle* events are prohibited when using Yakindu.

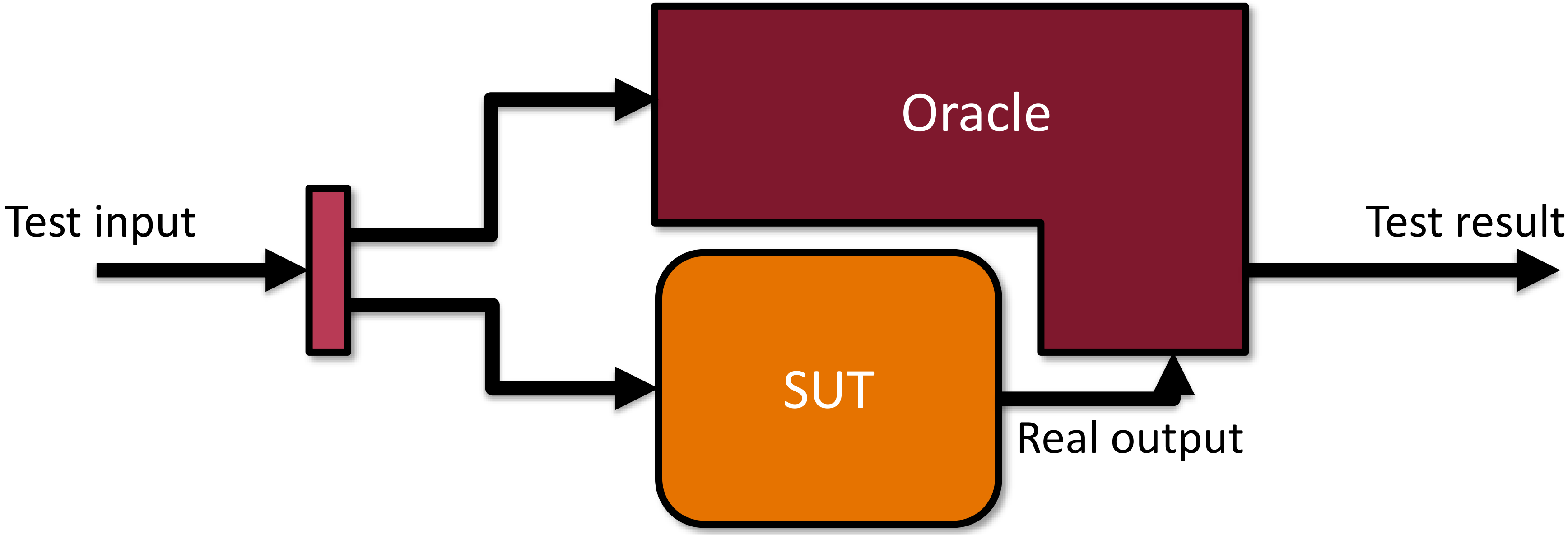
TESTING

Model Testing



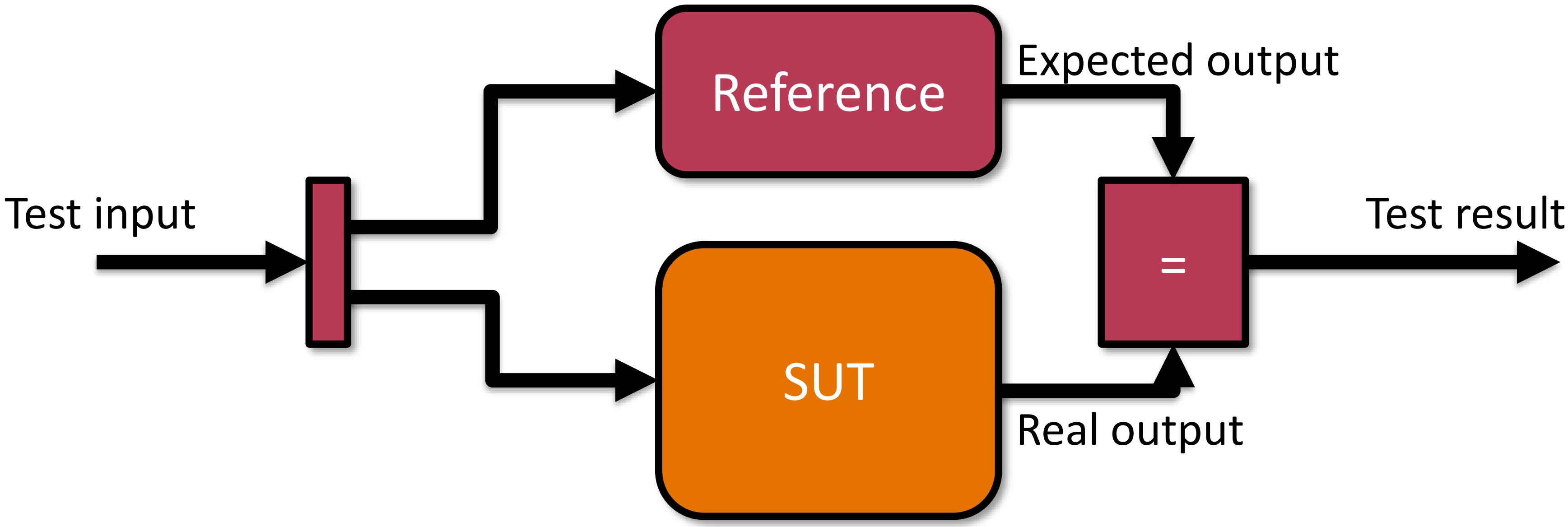
Model Testing

- **Oracle:**
  producing and comparing expected results

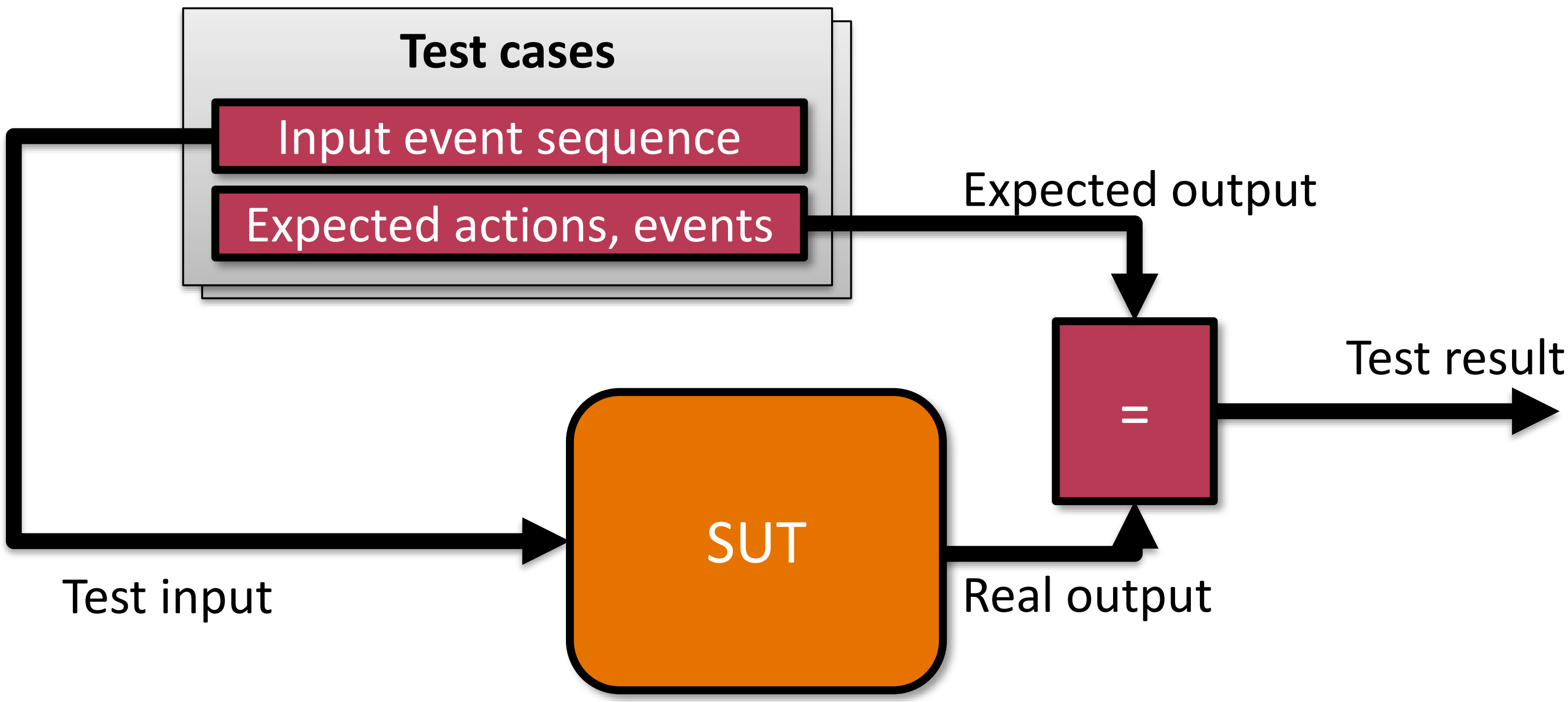


Model Testing

- **Reference:**
  expected output based on test input



Model Testing Example: State Machine



Model Testing Example: Yakindu State Machine

Example test case: In Settings menu, the initial time can be set between 1 and 3 minutes on a 5 seconds scale.

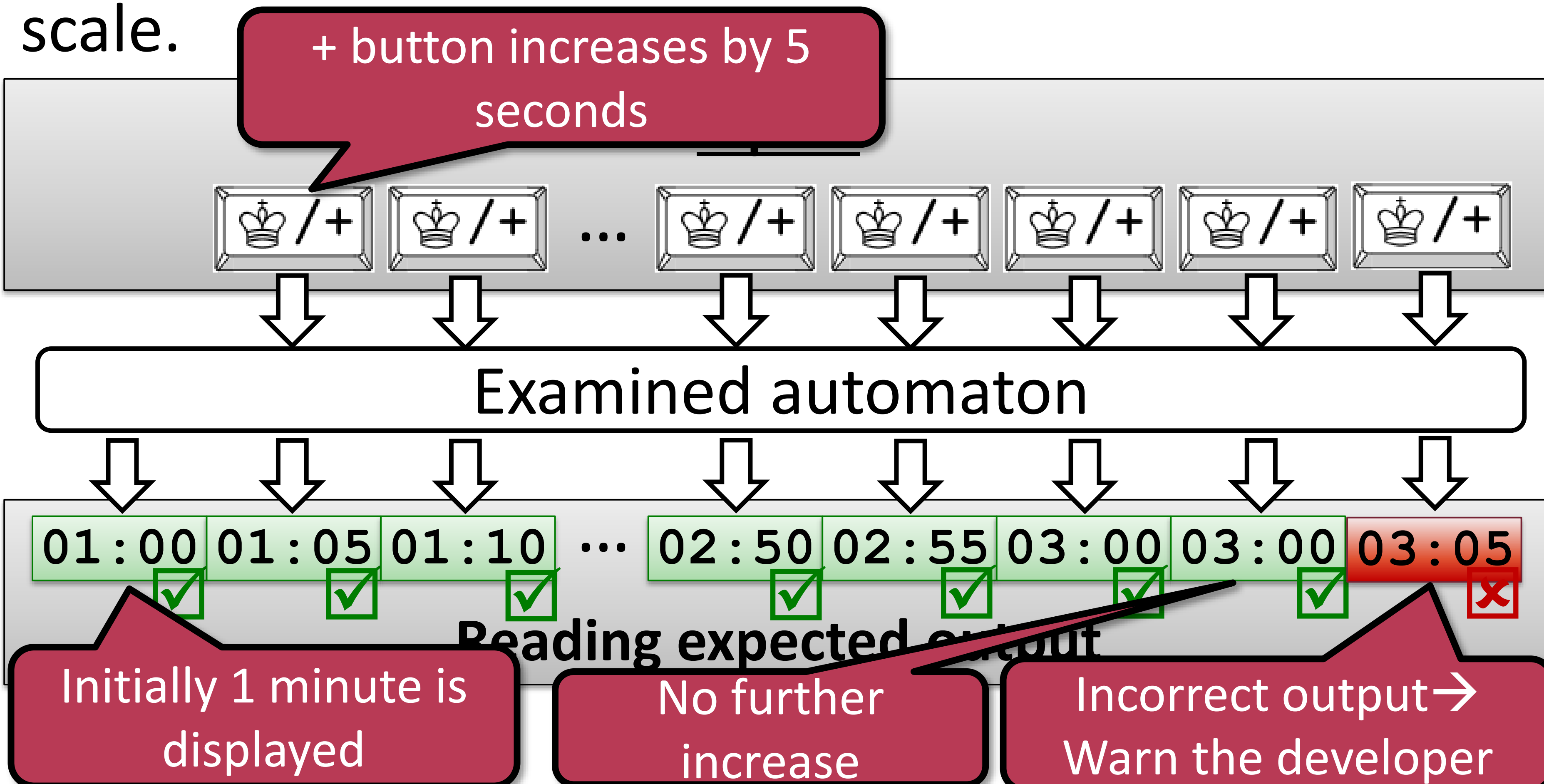
Inputs

Examined automaton

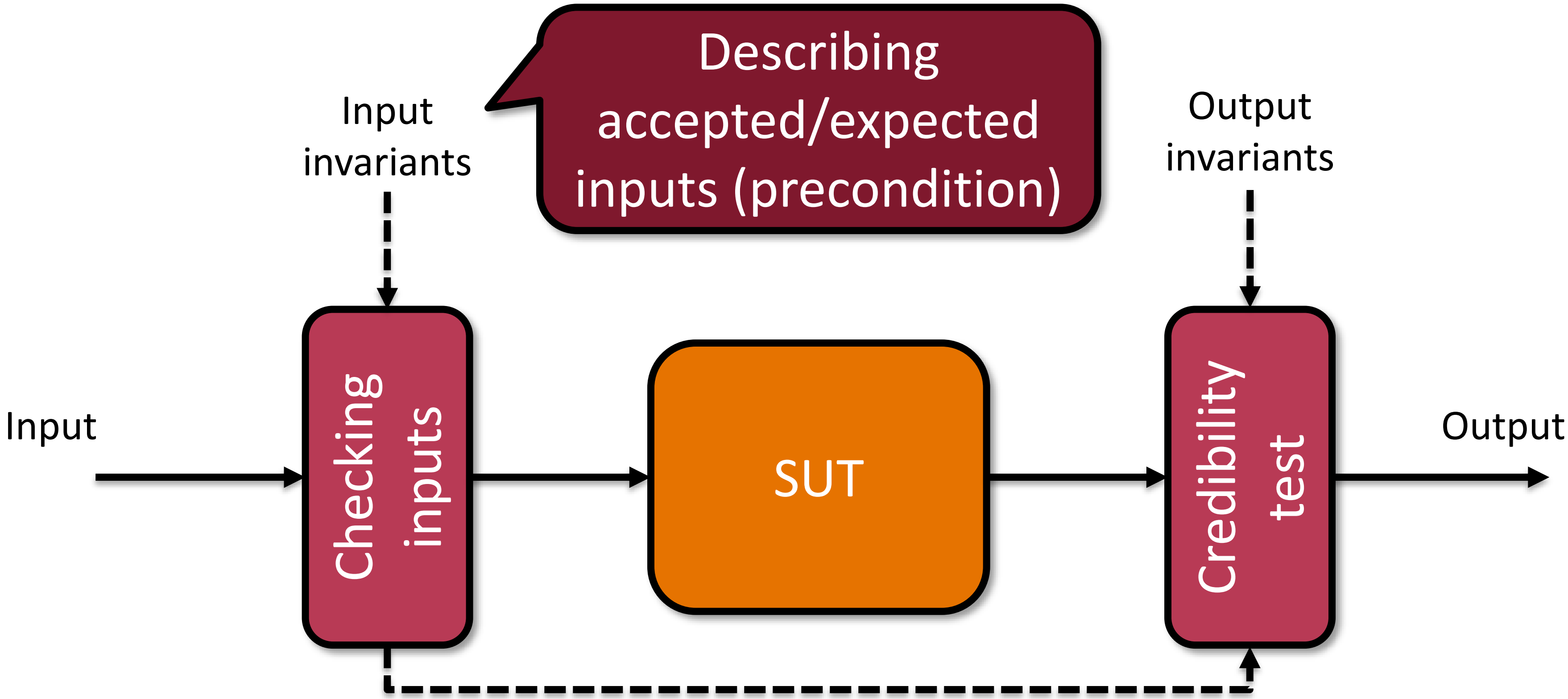
Reading expected output

Model Testing Example: Yakindu State Machine

Example test case: In Settings menu, the initial time can be set between 1 and 3 minutes on a 5 seconds scale.

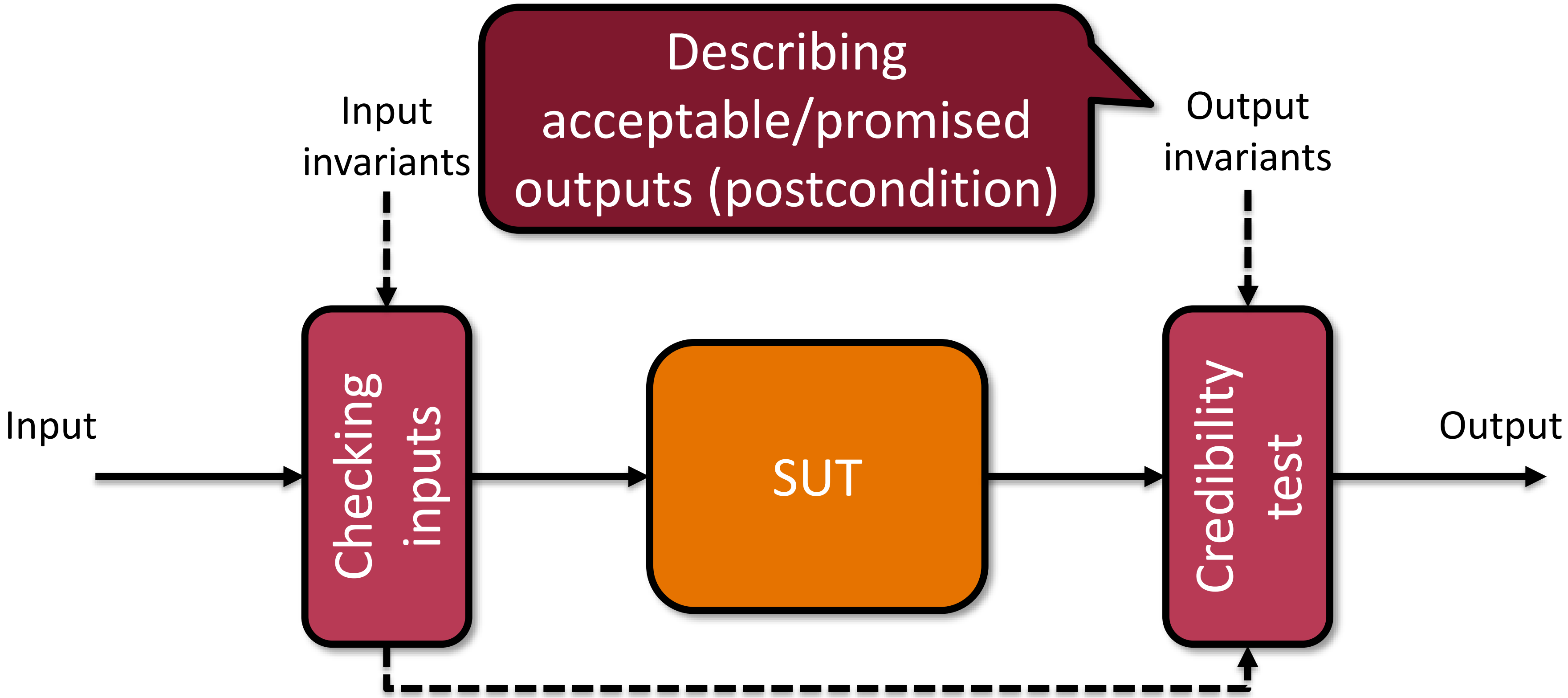


Self Testing (Monitor)



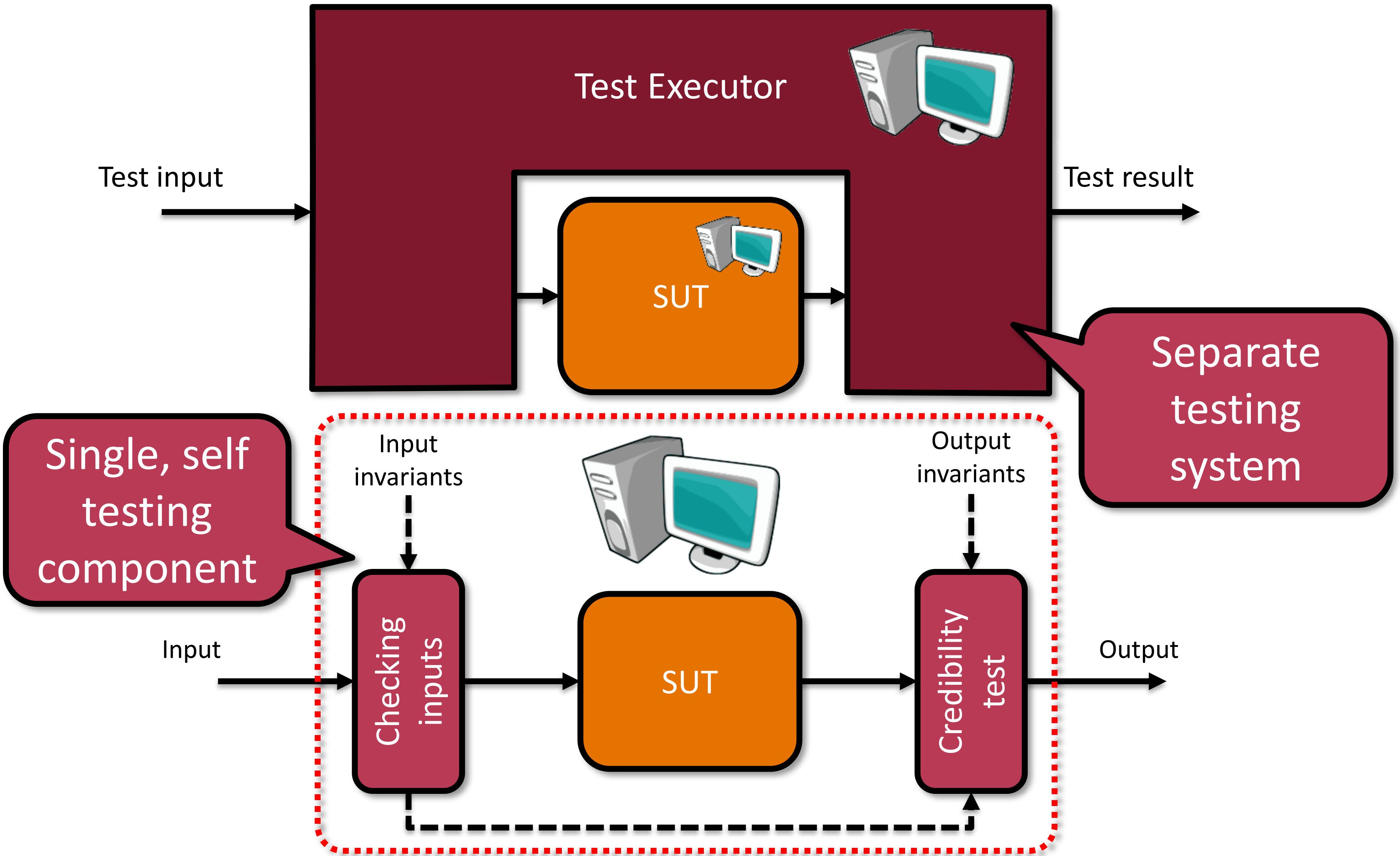
- **Invariant property:**
  must be continuously true

Self Testing (Monitor)



- **Invariant property:**
  must be continuously true

Self Testing vs. External Testing



Self Testing Program

Pre-condition: discriminant is non-negative

```
void Roots(float a, b, c,
           float &x1, &x2)
{
    float d = sqrt(b*b-4*a*c);

    x1 = (-b + d)/(2*a);
    x2 = (-b – d)/(2*a);
}
```

Post-condition: both solutions are zero

```
void RootsMonitor(float a, b, c,
                  float &x₁, &x₂)
{
    //precondition
    float D = b²-4·a·c;
    if (D < 0)
        throw "Invalid input!";

    // execution
    Roots(a, b, c, x₁, x₂);

    // postcondition
    assert(a·x₁²+b·x₁+c == 0 &&
           a·x₂²+b·x₂+c == 0);
}
```

Self Testing Program

Prec... ...ve

```
voi...

{
    float d = sqrt(b*b-4*a*c);

}
```

P...

Exception:
Unexpected situation, differing from normal.
Handling is implemented at some other part.

Reason: misuse.

Assert (presumption):
Erroneous state, that the code **isn't prepared to handle**.

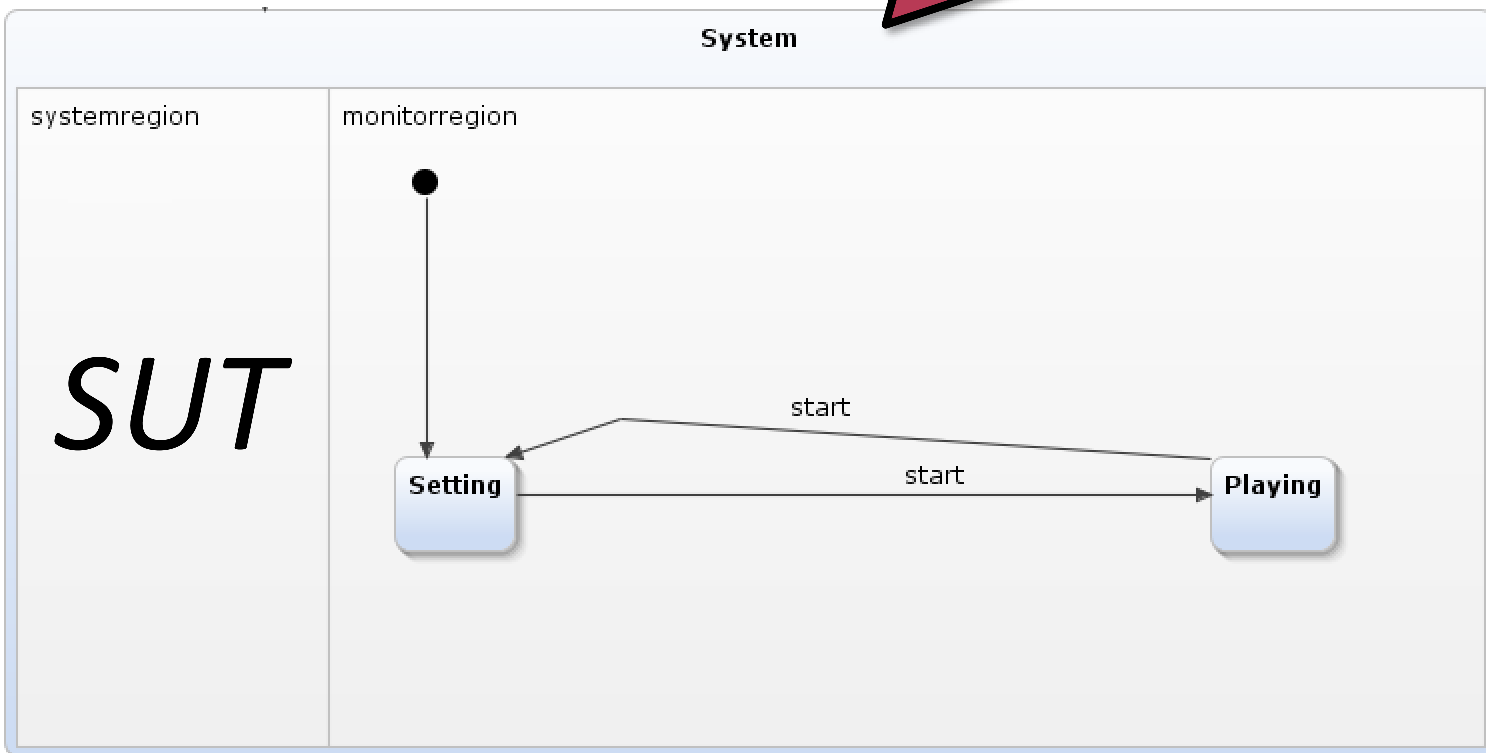
Reason: incorrect implementation or runtime error.

```
void RootsMonitor(float a, b, c,
                  float &x1, &x2)
{
   //precondition
   float D = b²-4·a·c;
   if (D < 0)
      throw "Invalid input!";

   // execution
   Roots(a, b, c, x1, x2);

   // postcondition
   assert(a·x1²+b·x1+c == 0 &&
          a·x2²+b·x2+c == 0);
```

Monitoring in Yakindu

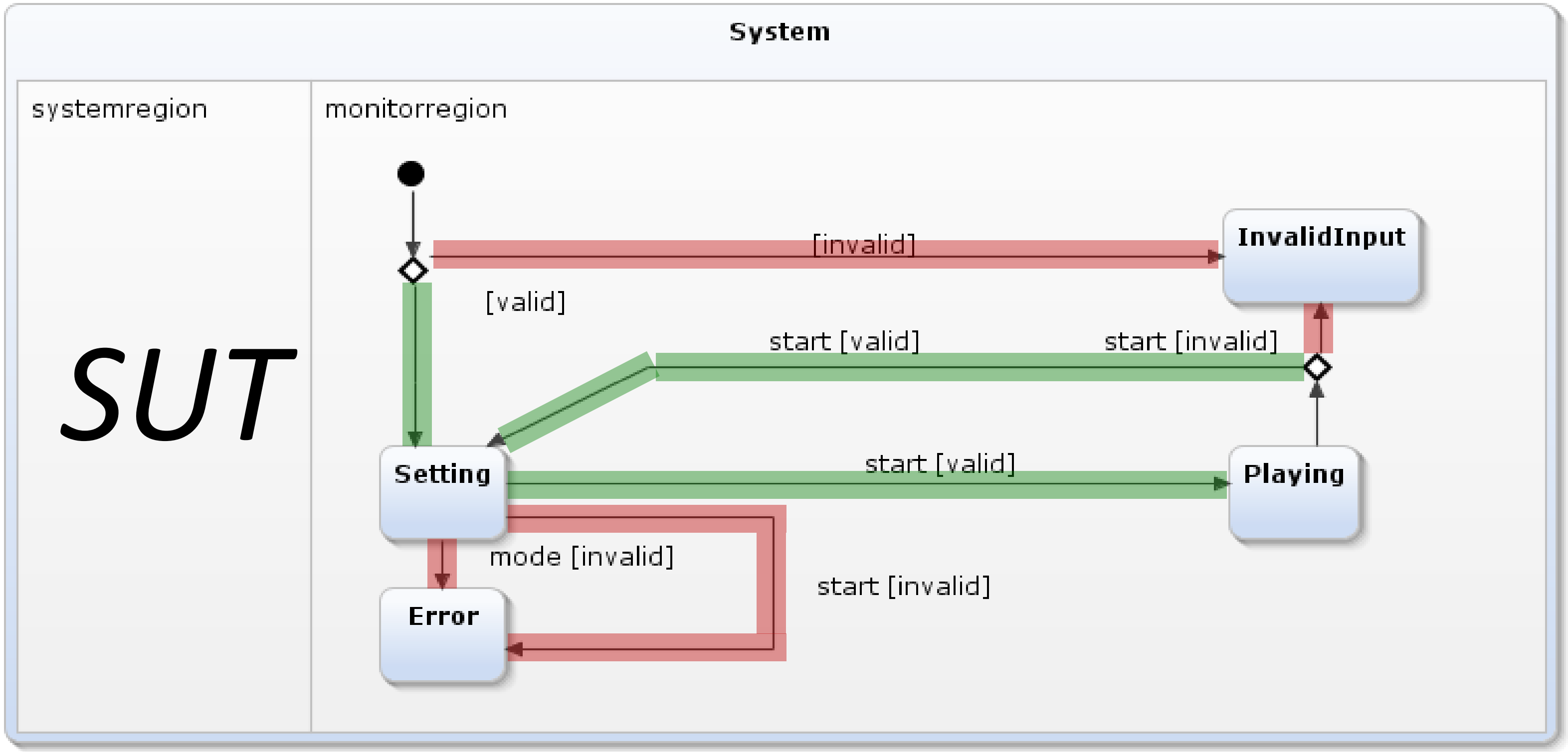
- *SUT* and *monitor* regions running paralelly
  - Good case:
    - Valid input
    - Correct operation

In the homework, one can switch between setting and playing.



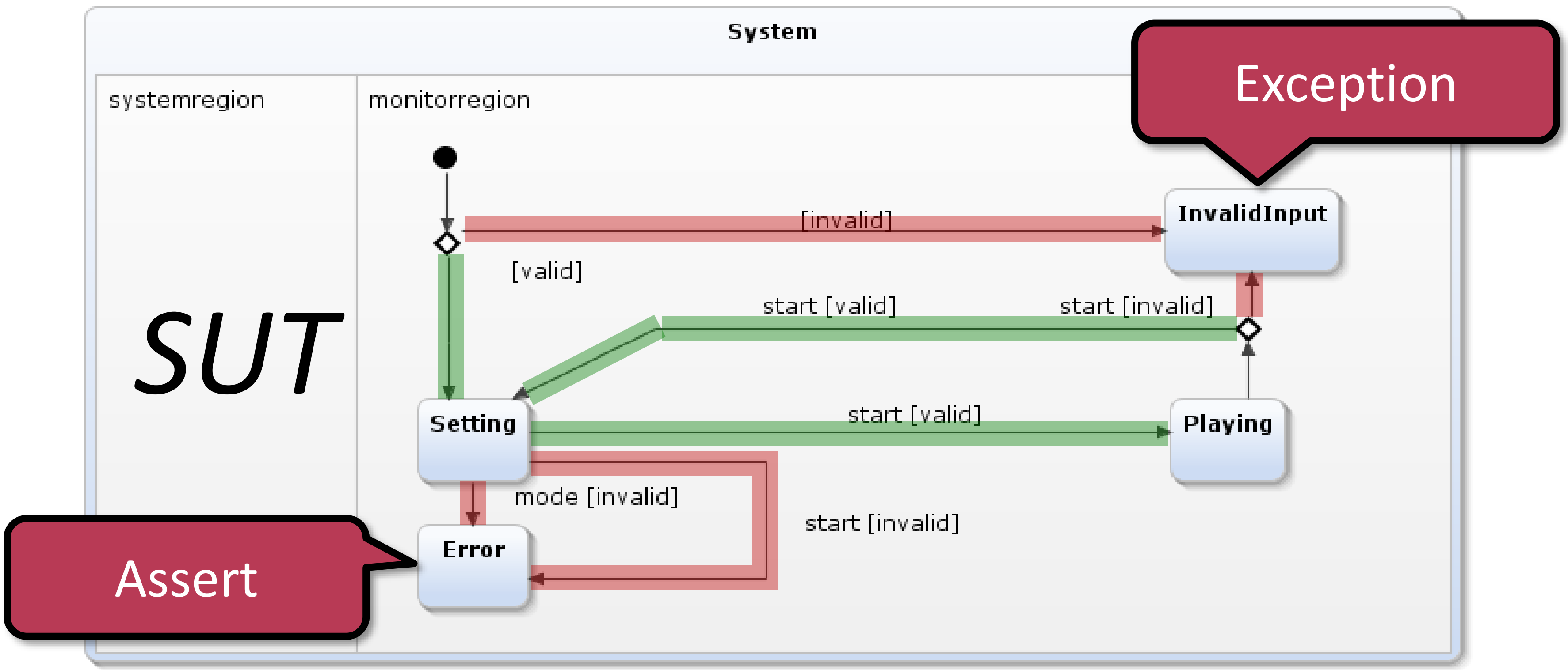
Monitoring in Yakindu

- *SUT* and *monitor* regions running concurrently
  - Good case:
    - Valid input
    - Correct operation
  - Bad case:
    - Invalid input → InvalidInput
    - Incorrect output → Error



Monitoring in Yakindu

- *SUT* and *monitor* regions running parallelly
  - Good case:
    - Valid input
    - Correct operation
  - Bad case:
    - Invalid input → InvalidInput
    - Incorrect output → Error



Model Testing

- Executing the model: Simulation
  - Analysing behaviour for given inputs
- Test case:
  1. Test input
     - e.g. a mid-range value and two corner cases
  2. Expected output

What inputs should be tested?

Coverage

- **Coverage** is the ratio of concerned model parts during the execution of a given test suite.
  - State coverage (in state machines):

$$\frac{\textbf{reached states}}{\textbf{all states}}$$

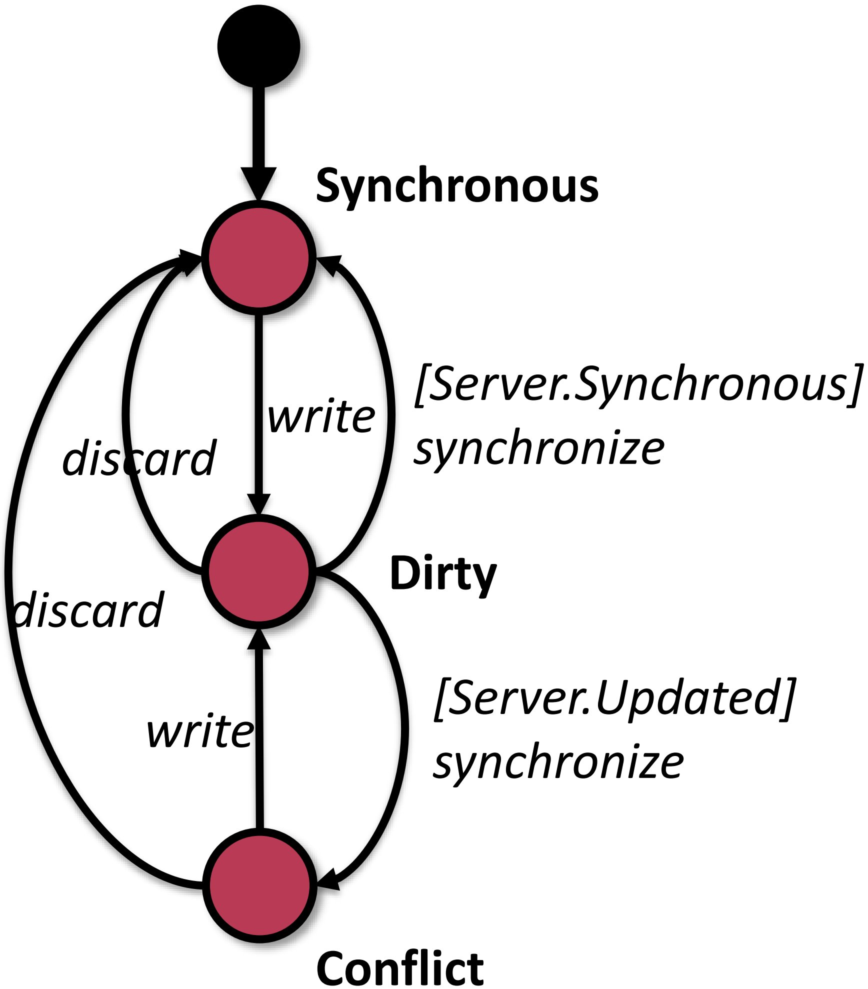
  - Transition coverage (in state machine):

$$\frac{\textbf{fired transitions}}{\textbf{all transitions}}$$

  - Command coverage (in control flow):

$$\frac{\textbf{executed activities}}{\textbf{all activities}}$$

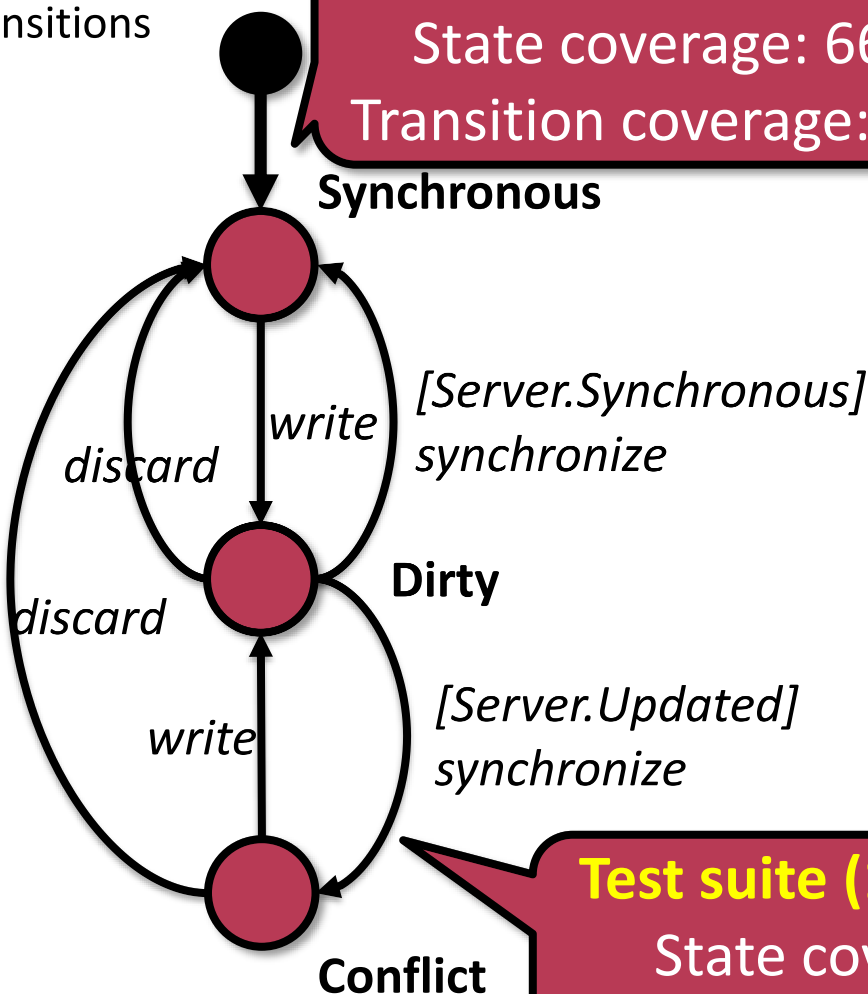
Example: Cloud-based Data Storage



„We are modelling cloud based data storage with only one file. The client can write the file, synchronize with the server and discard local modifications. Depending on the version of the replica on the server synchronizing may cause conflict if others have modified the file."

Example: Cloud-based Data Storage

3 states
6 transitions



Test suite coverage:
State coverage: 66%
Transition coverage: 33%

Test case:

a) write
b) discard

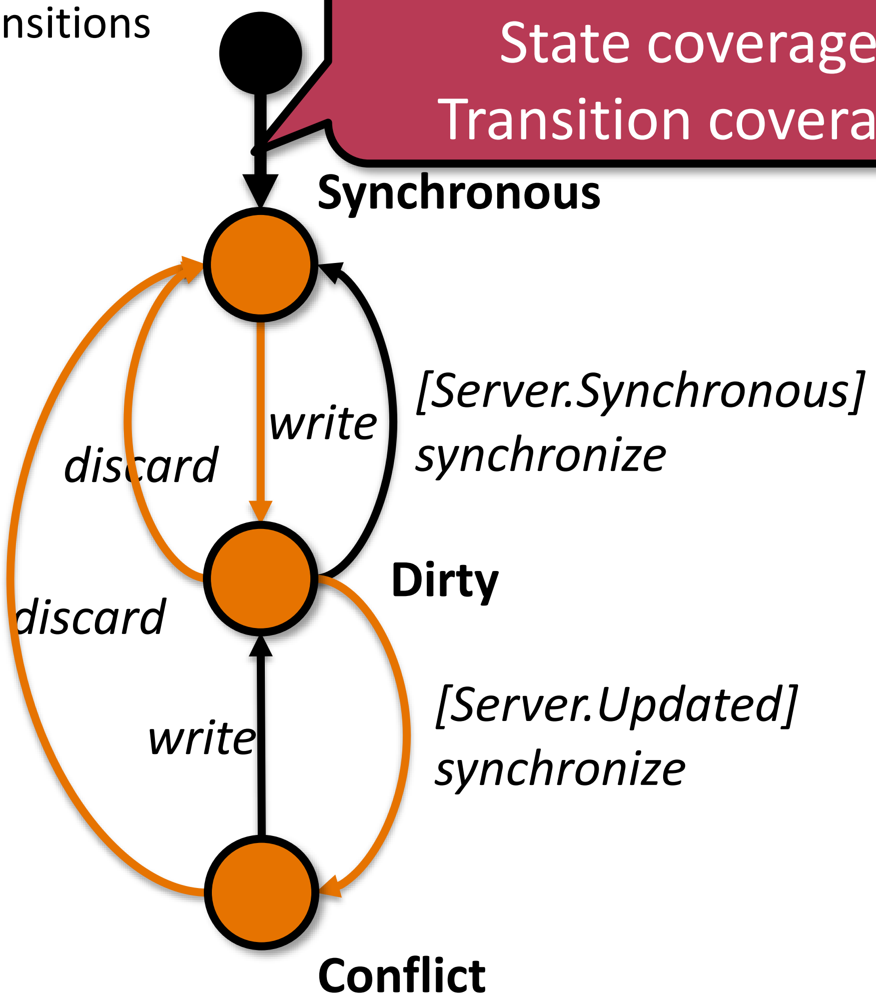
2. Test case:

a) write
b) Server = Updated
c) synchronize
d) discard

Test suite (1.+2.) coverage:
State coverage: 100%
Transition coverage: 66%

Example: Cloud-based Data Storage

3 states
6 transitions



Test suite (1.+2.+3.) coverage:

State coverage: 100%

Transition coverage: 100%

…ase:

…e

b) Server = Updated

c) synchronize

d) write

e) Server = Synchronous

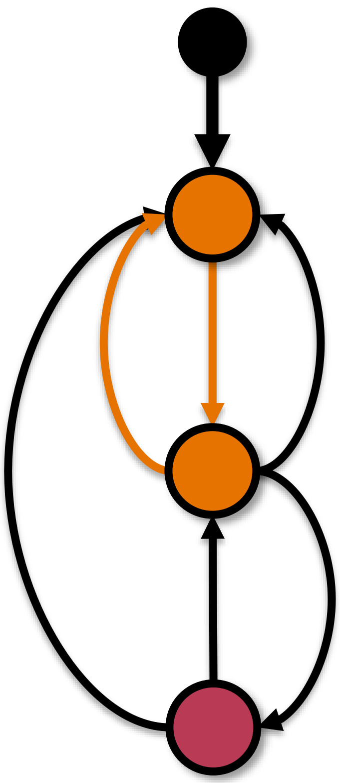
f) synchronize

Coverage

After first test case:

State coverage: 2/3=66%

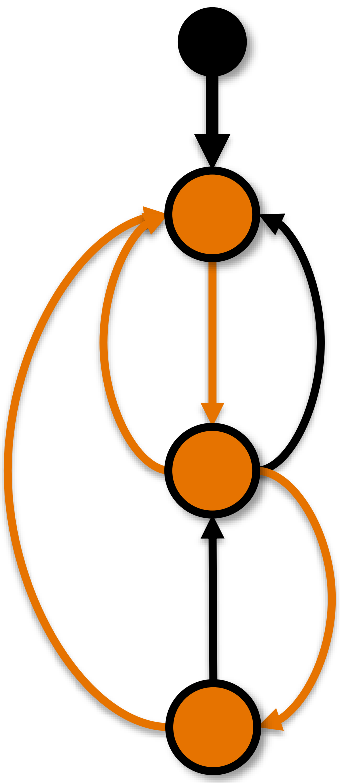
Transition coverage: 2/6=33%



After second test case:

State coverage: 3/3=100%

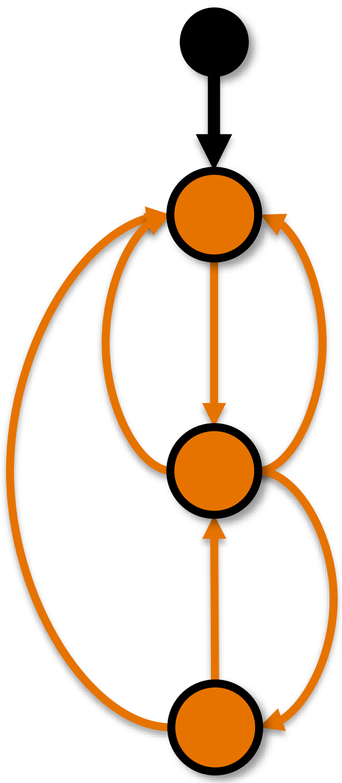
Transition coverage: 4/6=66%



After third test case:

State coverage: 3/3=100%

Transition coverage: 6/6=100%



Using Tested Models

- **Software testing:**
  - Reusing (100% coverage) test suite
  - Covering test inputs (input)
  - Outputs by model (expected output)
- **Monitoring:** simulating the model while running the software
  - Same inputs for the model and the program
  - Comparing outputs → **fault detection**
- **Log analysis:**
  - Running the monitor over logged input/outputs

Using Tested Models

- **Software testing:**
  - Reusing (100% coverage) test suite
  - Covering test inputs (input)
  - Outputs by model (expected output)
- **Monitoring:** simulating the model while running the software
  - Same inputs for the model and the p
  - Comparing outputs → **fault detectio**
- **Log analysis:**
  - Running the monitor over logged inp

Before running

While running

After running

Test Documentation

- Test cases and test results should be documented!

Test specification

  - What does it test?
  - Based on what requirement?
  - What is the input?
  - What outputs are expected?

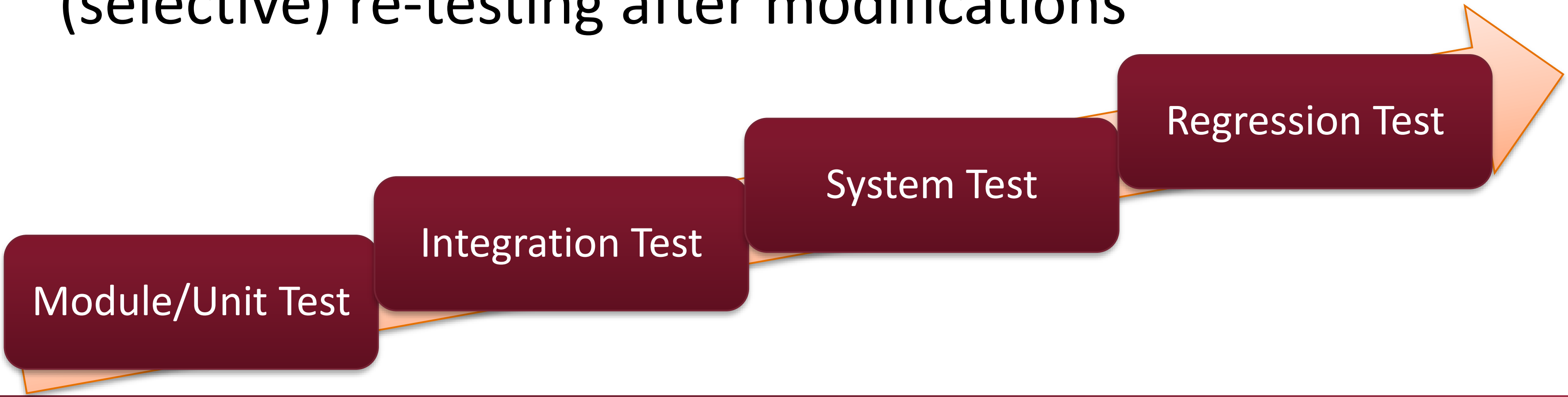
Test report

  - Has it been executed?
  - If so, was it succesful?

- Traceability:
  - Exploring untested code lines and unsatisfied requirements
  - Recording and tracing back the test results

Types, Phases of Tests

- **Module testing:**
  separating and testing a component
- **Integration test:**
  testing multiple components together
- **System test:**
  testing the complete system together
- **Regression test:**
  (selective) re-testing after modifications



FORMAL VERIFICATION

Formal Verification

- **Formal verification:** proving correctness of models/programs with mathematical methods
  - For more information see: Formal Methods masters course
- Tools:
  - **Model checking**
    - Exhaustive examination of possible behaviours
  - Automatic proof of correctness
    - Automatic theorem proving based on axiom systems
  - Conformance testing
    - Checking compatibility between models

Model Checking

- **Model checking:** exhaustive (complete) analysis of possible behaviour of the model, based on given requirements
  - Search for erroneous operation

    → **Counter example**

Testing	**Model Checking**
Small set of possible cases	Complete
Checks expected outputs	Checks a sequence of states
Requires less computation	Requires more computation
Does not prove correctness	Proves formally