Model Verification and Validation

Course Leader: Dr. András Pataricza
Lecturer: Gábor Guta

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
  - Primary reason:
    Unsuccessful conversion between 64 bit and 16 bit number
  - Secondary reason:
    Modules were never tested together
Example: Esterel SCADE

**SCADE Suite**

- **Specification**
- **Models**
- **Verification**
- **Generation**

**PROTOTYPE & DESIGN**

**VERIFY**

- Model Checking
- Formal Verification
- Debug & Simulation
- Rapid Prototyping & Executable Spec
- Model Coverage Analysis
- Time & Stack Analysis

**GENERATE**

- SCADE Suite KCG
  - C & Ada
- RTOS Adaptors
- Object Code & Compiler Verification
- DO-178B
- DO-178C
- IEC 61508
- EN 50128
- ISO 26262
- Certification Kits

**Plane, critical ES, car**
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

Model development
- Requirements, specification
- Initial models
- Detailed models
- Verification
- Implementation
- Testing
- Maintenance

Software development
- Requirements, specification
- Implementation
- Verification
- Testing
- Maintenance

Correct Model, Correct Code
Basic concepts

Static analysis

Testing

Formal verification

BASIC CONCEPTS
Models and Activities

- **Synthesis:**
  
  Model conformant to specification?

- **Analysis:**
  
  Model’s behavior?

- **Control:**
  
  How can desired state be reached?
Correctness

- **Correctness:**
  - model/code fulfills the requirements.
    - **Functional Correctness:** satisfying the functional requirements.
    - Checking non-functional requirements: see lecture on Performance modeling

- **Aspects:**
  - Always able to complete the task
  - Error-free
  - No forbidden behavior
Classification of Functional Requirements

- **Safety properties** *(allowed behavior):*
  - „something bad will never happen”
  - What state can/can’t be the state of the system
  - What behavior is prohibited
  - Universal requirements
    - They must always be true

- **Liveliness properties** *(expected behavior):*
  - „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
Classification of functional requirements

- **Allowed** behavior:
  - What state can/can’t be the state of the system
  - What behavior is prohibited
  - Universal requirements
    - They must always be true

- **Expected** behavior:
  - What states should be able to be reached
  - What functions should the system be capable of
  - Existential requirements
    - Possibility of fulfilling

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

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

- **Deadlock state:**
  - the system can only step out of it by external assistance
  - Eg. Processes waiting for each other
At crossroads – unless road signs or traffic rules tell otherwise – the vehicle coming from the right has 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 their priority, and let the other go. If they don’t they will stay there forever according to Highway code. (gyakorikerdesek.hu)
Unlocking the Deadlock

If 4 cars arrive to the intersection at the same time, then one of them has to disclaim their priority, and let the other go. If they don’t they will stay there forever according to Highway code. (gyakorikerdesek.hu)
Another Deadlock

- If 4 cars arrive to the intersection at the same time, then one of them has to disclaim their priority, and let the other go. If they don’t they will stay there forever according to Highway code. (gyakorikerdesek.hu)
Unlocking the Deadlock

- At the intersection – unless road signs or traffic rules tell otherwise – the vehicle coming from the right has priority. (Road Traffic Act I, 1988)
Unlocking the deadlock

- At the intersection – unless road signs or traffic rules tell otherwise – the vehicle coming from the right has priority.
  (Road Traffic Act I, 1988)

Real life: broken traffic lights at Belgrade Quay (all of them were red...)

Then I shall...
Infinite Loop (livelock)

- **Deadlock** *set of states*: the system can only step out of it by external assistance
  - Eg. Result of unlocking the deadlock
Deadlock

- **Deadlock:** the system can only step out of it by external assistance
  - Eg. Processes waiting for each other

- **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"
Deadlock

- Deadlock: the system can only step out of it by external assistance
  - Eg. Processes waiting for each other

- Common design mistakes in 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"

- "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

Model

DFN of the communication of the bank and the travel agency

Verification and validation

Requirement

„No message is lost and sooner or later the communication ends.”
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
    - Random like check (testing, simulation)
    - Complete (model checking)
Basic concepts

Static analysis

Testing

Formal verification

STATIC ANALYSIS

Error Patterns
Is the following model correct?
Decision and Join

- Is the following model correct?

- Join: only continues when tokens arrived from all inputs

→ DEADLOCK
Is the following model correct?
Fork and Merge

- Is the following model correct?

- Merge: let the token pass through from any branch
  - Doesn’t synchronize

→ „Destock” is executed twice
Cycle 1.

- Is the following model correct?
Cycle 1.

- Is the following model correct?

- Join: only continues when tokens arrived from all inputs

→ DEADLOCK
Cycle 2.

- Is the following model correct?
Cycle 2.

- Is the following model correct?

- Join: csak akkor léphet tovább, ha mindegyik bemenetén érkezett token

→ DEADLOCK
Cycle 3.

- Is the following model correct?
Cycle 3.

- Is the following model correct?
Cycle 3.

- Is the following model correct?

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

  **Borderline case...**
Cycle 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?
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
**Moral:** These incorrectnesses can be avoided by using well structured processes

Allowed patterns:
Static analysis of data management

- A process multiplies two numbers
  - Derived requirement:
    - „If at least one of them is even, the result will also be.”
  - Can be trace through the code
    - „Executing in mind”

- Symbolic execution
  - Instead of concrete values the program is executed with sets of values
  - Interesting inputs can be defined
    - Eg. Internal branches
      ➔ By what inputs can the branches be reached?
Static analysis: syntax analysis

- Syntax analysis: modeling 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 \Rightarrow Couldn’t resolve reference
  - Advanced editor (offering possibilities for instance)

- Code and diagram together

- Programming: incorrect during editing
  Modeling: correct during editing
Static analysis: structural correctness

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

![Diagram showing a model graph with a node: RoundWhite connected to a Paused state, with a Node is not reachable annotation.]

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

- **Example 3.** Supporting design guidelines: Further rules can be added to the model.
  
  - *Always* and *Oncycle*: Event firing on clock tick
  - Arbitrary frequency → Typically malfunction

**Using *Always* and *Oncycle* events are prohibited in the home assignment.**
TESTING

Just trial and error?
Model testing

Test executor

Test input → SUT → Test result

System Under Test
- **Oracle:**
  producing and comparing expected results

![Diagram](https://via.placeholder.com/720x540.png)

- **Model testing**
  - **Oracle:** producing and comparing expected results
  - **SUT:**
  - **Test input**
  - **Test output**
  - **Real output**
- **Reference:**
  expected output based on test input

---

**Model testing**

1. **Reference**: expected output based on test input
2. **SUT**: System Under Test
3. **Test input**: Input for the test
4. **Real output**: Output from the SUT
5. **Expected output**: Expected output based on the test input
6. **Test result**: Result of the test, comparing the Real output with the Expected output

Diagram:
- Test input flows to the Reference block, which generates the Expected output.
- The Expected output is compared with the Real output from the SUT.
- The Test result indicates whether the SUT produced the expected output.
Model testing example: Yakindu state machine

- **Test cases**
  - Input event sequence
  - Expected actions, events

- **SUT**
- **Test input**
- **Real output**
- **Expected output**
- **Test result**
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 input
Examined automaton

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

Initially 1 minute is displayed

No further increase

Incorrect output ➔ Warn the developer

+ button increases by 5 seconds
Self testing (monitor)

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

Diagram:
- **Input** invariants to **Checking inputs**
- **SUT**
- **Credibility test**
- **Output** invariants

Describing accepted/expected inputs (precondition)
Self testing (monitor)

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

Describing acceptable/promised outputs (postcondition)

<table>
<thead>
<tr>
<th>Input invariants</th>
<th>Checking inputs</th>
<th>SUT</th>
<th>Credibility test</th>
<th>Output invariants</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td></td>
<td></td>
<td>Output</td>
<td></td>
</tr>
</tbody>
</table>
Self testing vs. External testing

Test Executor

Test input → SUT → Test result

Input invariants → Output invariants

Checking inputs → SUT → Credibility test

Single, self testing component

Separate testing system
Self testing program

Precondition: discriminant is non-negative

```c
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);
}
```

Postcondition: both solutions are zero

```c
void RootsMonitor(float a, b, c,
    float &x1, &x2)
{
    //precondition
    float D = b^2-4.a.c;
    if (D < 0)
        throw "Invalid input!";

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

    // postcondition
    assert(a*x1^2+b*x1+c == 0 &&
           a*x2^2+b*x2+c == 0);
}
```
Self testing program

**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.

```c
void RootsMonitor(float a, b, c, float &x, &x)
{
    //precondition
    float D = b^2 - 4*a*c;
    if (D < 0)
        throw "Invalid input!";
    // execution
    Roots(a, b, c, x, x);
    // postcondition
    assert(a*x^2 + b*x + c == 0 &&
           a*x^2 + b*x + c == 0);
}
```
Monitoring in Yakindu

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

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

- **SUT and monitor regions running parallely**
  - **Good case:**
    - Valid input
    - Correct operation
  - **Bad case:**
    - Invalid input $\rightarrow$ InvalidInput
    - Incorrect output $\rightarrow$ Error
Monitoring in Yakindu

- **SUT and monitor regions running paralelly**
  - **Good case:**
    - Valid input
    - Correct operation
  - **Bad case:**
    - Invalid input $\rightarrow$ InvalidInput
    - Incorrect output $\rightarrow$ Error
Model testing

- Executing the model: simulation
  - Analyzing behavior for given inputs

- Test case:
  1. Test input
    - Eg. Mid-range and two edges
  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):
    \[
    \frac{\text{reached states}}{\text{all states}}
    \]
  - Transition coverage (in state machine):
    \[
    \frac{\text{fired transitions}}{\text{all transitions}}
    \]
  - Command coverage (in control flow):
    \[
    \frac{\text{executed activities}}{\text{all activities}}
    \]
Example: Cloud-based data storage

„We are modeling 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

Test case:

1. Test case:
   a) write
   b) discard

2. Test case:
   a) write
   b) Server = Updated
   c) synchronize
   d) discard

3 states
6 transitions

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

Test suite coverage:
State coverage: 100%
Transition coverage: 66%
Example: Cloud-based data storage

Test suite coverage:
State coverage: 100%
Transition coverage: 100%

Test case:

a) write
b) Server = Updated
c) synchronize
d) write
e) Server = Synchronous
f) synchronize

3 states
6 transitions
Coverage

After first test case:
State coverage: $\frac{2}{3}=66\%$
Transition coverage: $\frac{2}{6}=33\%$

After second test case:
State coverage: $\frac{3}{3}=100\%$
Transition coverage: $\frac{4}{6}=66\%$

After third test case:
State coverage: $\frac{3}{3}=100\%$
Transition coverage: $\frac{6}{6}=100\%$
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 program
  - Comparing outputs → **fault detection**

- **Log analysis:**
  - Running the monitor over logged inputs/outputs

---

**Before running**

**While running**

**After running**
Test Documentation

- Test cases and test results should be documented!
  - What does it test?
  - Based on what requirement?
  - What is the input?
  - Has it been executed?
  - If so, was it successful?

- Traceability:
  - Exploring untested code lines and unsatisfied requirements
  - Recording test results
Types, phases of tests

- **Modul 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
Basic concepts
Static analysis
Testing

FORMAL VERIFICATION

Model 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 behaviors
  - **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 behavior of the model, based on given requirements
  - Search for erroneous operation
    - **Counter example**

<table>
<thead>
<tr>
<th>Testing</th>
<th>Model Checking</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small set of possible cases</td>
<td>Complete</td>
</tr>
<tr>
<td>Checks expected outputs</td>
<td>Checks a sequence of states</td>
</tr>
<tr>
<td>Requires less computation</td>
<td>Requires more computation</td>
</tr>
<tr>
<td>Does not prove correctness</td>
<td>Proves formally</td>
</tr>
</tbody>
</table>
SZÓSZEDET
<table>
<thead>
<tr>
<th>Magyar</th>
<th>English</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>