Control software for industrial automation systems

IEC 61499 – a standard for distributed control systems which supports modularity

Testing is one of the ways to ensure system correctness and safety

Automatic test suite generation

Optimization goal: coverage of the implementation

Unreachable system parts can be identified
Black-box and white-box testing

- **Model-based testing** is a black-box approach based on requirements
- The model of the requirements and the implementation are independent

- **White-box testing** is based on implementation instead
  - What if a short requirement has a complex implementation with many branches? – WB testing should be more effective
  - What if some requirements are not implemented at all? – only MBT will reveal this
Function blocks (FBs)

- Interface: event and data inputs and outputs
- FBs have state and behaviour
- Two types: basic (based on execution control charts, ECCs) and composite FBs
Function blocks as systems under test

- Each test represents a number of execution steps
- On each step, the FB receives an event and data (variable values) associated with it
- Each step also triggers some measurable output
- E.g. \((E3, \text{BOOL\_VAR} = \text{true}) \rightarrow (O1, \text{OV} = \text{false})\)
Tests and test suites

- A test is defined as a sequence of inputs; outputs are not considered and are not needed for coverage optimization.

- Test example:

<table>
<thead>
<tr>
<th>Event</th>
<th>BOOL_VAR</th>
<th>INT_VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>E3</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>false</td>
<td>-100</td>
</tr>
<tr>
<td>E2</td>
<td>false</td>
<td>42</td>
</tr>
</tbody>
</table>

- A test suite is a number of tests.

- The goal of the research is to design a method which generates test suites with high coverage.
Coverage criteria

In general, coverage determines the share of the system which is executed at least once when all tests are run.

For basic FBs:
- transition coverage of underlying ECCs
- branch coverage of the code generated from the FB

For composite FBs:
- aggregated measures of nested basic FBs

Coverage criteria can be viewed as percentage values (e.g. 50% transition coverage means that half of transitions were executed)
How to cover all 5 transitions of the execution control chart?
Transition coverage example (2)

Test 1: (E3, BV = true); (E1); (E2, BV = false, IV = 10)
Transition coverage example (3)

Test 2: \((E2, BV = true, IV = -1); (E1)\)
Test 3: (E2, BV = true, IV = -1); (E3)
Evolutionary computation: basic ideas

- General optimization methodology for both discrete and continuous problems

- One needs to define a representation of solutions (individuals)
  - E.g. a list of tests, where a test is a list of (event, data) pairs

- A **fitness function** evaluates the individuals
  - E.g. the coverage percentage according to a given criterion

- Evolutionary operators (e.g. **mutation, crossover**) modify the solutions

- Many algorithms (e.g. the genetic algorithm) with a common idea: discard worst solutions, retain better ones
Test suite optimization problem and its solution

- An FB to test (system under test) and a coverage criterion (fitness function) are defined
- Test suites (individuals) are evolved according to some particular evolutionary algorithm
- If a test suite with 100% coverage is found, the algorithm is stopped
- In not, the algorithm can still stop after several unsuccessful iterations

Note: no guarantees of optimality
An FB is transformed to source code, and then software for coverage-based unit test generation is applied.
Stage 1: FBDK execution

- FBDK is an FB development environment
- It can transform FBs into Java code
- I.e. there are variable and event declarations, methods for algorithms and events
- FBDK execution is implemented as a library call
Stage 2: Source transformations

- Mostly technical, not very important
- To prepare the code for EvoSuite
- Some method visibility adjustments
- Nested FB definitions are inlined into a composite FB
Stage 3: EvoSuite execution

- A tool which generates JUnit coverage test suites for Java code
- EvoSuite implements the steady-state genetic algorithm (and some others, but these options were not considered)
- The whole test suite is an individual

- 10 minutes of evolutionary search for basic FBs
- 20 minutes for composite FBs
## Tests as code

<table>
<thead>
<tr>
<th>Event</th>
<th>BOOL_VAR</th>
<th>INT_VAR</th>
</tr>
</thead>
<tbody>
<tr>
<td>E3</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>E1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>E2</td>
<td>false</td>
<td>-100</td>
</tr>
<tr>
<td>E2</td>
<td>false</td>
<td>42</td>
</tr>
</tbody>
</table>

ExampleFB fb = new ExampleFB();
fbb.service_E3(true);
fbb.service_E1();
fbb.service_E2(false, -100);
fbb.service_E2(false, 42);
System under test: pick-and-place manipulator (PnP)
System under test: heat production process (HPP)
Coverage value statistics for 43 basic and 18 composite FBs from the PnP and HPP systems

<table>
<thead>
<tr>
<th>FB type, coverage criterion</th>
<th>Min</th>
<th>First quartile</th>
<th>Median</th>
<th>Third quartile</th>
<th>Max</th>
</tr>
</thead>
<tbody>
<tr>
<td>Basic, branch</td>
<td>60.0%</td>
<td>88.3%</td>
<td>92.6%</td>
<td>94.8%</td>
<td>98.8%</td>
</tr>
<tr>
<td>Composite, branch</td>
<td>35.4%</td>
<td>79.5%</td>
<td>84.5%</td>
<td>91.0%</td>
<td>94.8%</td>
</tr>
<tr>
<td>Basic, transition</td>
<td>55.6%</td>
<td>100.0%</td>
<td>100.0%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
<tr>
<td>Composite, transition</td>
<td>5.7%</td>
<td>92.0%</td>
<td>100.0%</td>
<td>100.0%</td>
<td>100.0%</td>
</tr>
</tbody>
</table>
Results: found faults

- Badly written ‘if’ decisions, which made algorithm branches inaccessible in two basic FBs
- A forgotten update of an internal variable, which made several states of an ECC inaccessible
- Missing connections in a composite FBs
- Unused algorithms in several FBs
Example of an uncovered code fragment

```java
public void alg_REQ() {
    
    if (((DIH.value & DIL.value)
        | (AI.value >= PRESET_HH.value & !DIH.value)
        | (AI.value <= PRESET_LL.value & !DIL.value)
        | (AI.value > PRESET_MAXAI.value) | (AI.value <

    SensorFault.value = true;
    AlarmHH.value = false;
    AlarmH.value = false;
    AlarmL.value = false;
    AlarmLL.value = false;

    } else {
        SensorFault.value = false;

        if (AI.value >= PRESET_HH.value) {
            AlarmHH.value = true;
            AlarmH.value = false;
            AlarmL.value = false;
            AlarmLL.value = false;
            
        } else if (AI.value < PRESET_H.value & AI.value >= PRESET_H.value) {
            
            AlarmHH.value = false;
            AlarmH.value = true;
            AlarmL.value = false;
            AlarmLL.value = false;
            
        } else if (AI.value <= PRESET_L.value
```
Future work

- Exact solution of the problem (under several assumptions):
  - Symbolic (dynamic) execution and constraint satisfiability
  - Coverage of net/condition event systems (NCES) using formal verification
- Does white-box test generation indeed complement model-based testing?
- Adding restrictions on possible tests to prevent unrealistic ones
- Manual implementation of the evolution and the FB translation into code (already done)
Thank you for your attention!

Questions?