Testing Distributed Systems

Part II: Test Cases, Model Coverage and Requirements
Tracing — Coverage Measures for Distributed Systems
2012-08-01

Jan Peleska and Wen-ling Huang
University of Bremen
{jp,huang}@informatik.uni-bremen.de
Development Model (e.g. Simulink)

SysML Test Model

Manual derivation of test model from software requirements, developer model and hand-written code – performed during development model review
Requirements tracing to model elements – SysML method

REQ-CRASH-FLASHING-0003
Pressing and releasing one of the emergency flash switches de-activates crash flashing

REQ-CRASH-FLASHING-0004
Unlocking the doors via remote control de-activates crash flashing

CtrlCrashFlashing sets the cr_CrashFlashing output to 1 as soon as a crash impact has been signalled or has performed a 0 -> 1 -> 0 transition. Setting cr_CrashFlashing to 1 immediately blocks the normal crash flashing has command of the emergency flash switch.

There are 3 ways to de-activate crash flashing:
1. Pressing and releasing the emergency flash switch on the dashboard (db_EmsSwitch, state machine CRASH_FLASHING_ACTIVE -> EM_SWITCH_PRESSED -> CRASH_FLASHING_PASSIVE) Obviously activated by the release-funk, while emergency flashing can be both activated and de-activated by the state machine NormalAndEmerFlash:HandleCrashOverride)
2. Switching the special purpose emergency switch ON and OFF.
3. Opening the doors via central locking system.
Automated generation of model coverage test cases

TEST-CASE-CRASH-FLASHING-0017

Transition EM_SWITCH_PRESSED → CRASH_FLASHING_PASSIVE is performed correctly

TEST-CASE-CRASH-FLASHING-0018

Transition EM_SWITCH_SPV_PRESSED → CRASH_FLASHING_PASSIVE is performed correctly

TEST-CASE-CRASH-FLASHING-0019

Transition CRASH_FLASHING_ACTIVE → CRASH_FLASHING_PASSIVE is performed correctly

CtrlCrashFlashing sets the cr_CrashFlashing output to 1 as soon as a crash impact has been signaled. If the cr_CrashFlashing has performed a 0 → 1 → 0 transition, setting cr_CrashFlashing to 1 immediately blocks the normal crash-flashing, forcing command of the emergency flash switch.

There are 3 ways to de-activate crash-flashing:

1. Pressing and releasing the emergency flash switch on the dashboard (db_EmsSwitch, state machine CRASH_FLASHING_ACTIVE ➔ EM_SWITCH_PRESSED ➔ CRASH_FLASHING_PASSIVE). This activation is handled by the release-flash, while emergency flashing can be both activated and de-activated by a state machine NormalAndEmerFlash:HandleCrashOverride).
2. Switching the special purpose emergency switch ON and OFF.
3. Opening the doors via central locking system.
Automated tracing from test cases to requirements

<table>
<thead>
<tr>
<th>Requirement</th>
<th>Tested by</th>
</tr>
</thead>
<tbody>
<tr>
<td>REQ-CRASH-FLASHING-0003</td>
<td>TEST-CASE-CRASH-FLASHING-0017</td>
</tr>
<tr>
<td></td>
<td>TEST-CASE-CRASH-FLASHING-0018</td>
</tr>
<tr>
<td>REQ-CRASH-FLASHING-0004</td>
<td>TEST-CASE-CRASH-FLASHING-0019</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Automated tracing from test cases to development model components

<table>
<thead>
<tr>
<th>Simulink Component</th>
<th>Tested by</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulink.Crash_flashing. component_x</td>
<td>TEST-CASE-CRASH-FLASHING-0017</td>
</tr>
<tr>
<td></td>
<td>TEST-CASE-CRASH-FLASHING-0018</td>
</tr>
<tr>
<td></td>
<td>TEST-CASE-CRASH-FLASHING-0019</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Test Cases for Model Coverage

- Automatically identified in the test model
- Guaranteed to be “sufficient” according to requirements from RCTA DO178B/C, EN50128, IEC 26262, if
  - 100% decision coverage is achieved for non-critical code
  - 100% MC/DC coverage is achieved for safety-critical code
  - 100% requirements coverage
  - Test suite strength is sufficient
User-Defined Test Cases

Basic Control State & Interface Coverage

Hierarchic Transition Coverage

Transition Coverage

Basic Control State & Interface Coverage

Hierarchic MC/DC Coverage

Hierarchic MC/DC Coverage
Pairwise Testing with Orthogonal Arrays

Equivalence Class Partition Testing

User-Defined Test Cases

Basic Control State & Interface Coverage

Hierarchic Transition Coverage

Transition Coverage

Basic Control State Pairs Coverage

MC/DC Coverage

Hierarchic MC/DC Coverage

Pairwise Testing with Orthogonal Arrays
Equivalence Class Partition Testing

- Fundamental idea: input data processed in the SUT by
  - the same control path
  - the same algorithm

may be regarded as equivalent
Justification of equivalence class tests

• Equivalence class testing partitions the computation space restricted to SUT inputs, such that it may be expected that the SUT behaves “equivalently” for different members of each partition, in the following sense:

\[ \text{If 2 elements } x_o \text{ and } x_1 \text{ are members of the same partition (} = \text{equivalence class)} \text{, it may be expected that every error uncovered by } x_0 \text{ will also be uncovered by } x_1 \]
Equivalence Class Partition Testing

• **Example.** Input parameter *Voltage* in the turn indication example

• **Note.** It may be much more complex to “find the right” equivalence classes

☞ See last session in the afternoon

“Part VI: Abstraction and its Implication for Equivalence Testing”
Equivalence Classes in the Time Domain

Initial

Idle
- entry / lr_FlashCmd = 0;
- [b2_TurnIndLvr > 0]

Active
- entry / lr_FlashCmd = b2_TurnIndLvr;
- entry / lr_TipFlashing = 1;
- entry / t.reset();
- [b2_TurnIndLvr == 0]

Stable
- entry / lr_TipFlashing = 0;
- [t.elapsed(440)]

TipFlashing
- [b2_TurnIndLvr > 0]

- [t.elapsed(1980)] /lr_TipFlashing = 0;
3 Types of Equivalence Classes

- **Input equivalence classes** identify computations by means of their restriction to SUT inputs
- **Output equivalence classes** identify computation by means of restrictions to SUT outputs
- **Structural equivalence classes** identify computations covering similar parts (in general path segments) of the SUT code or SUT model
How path coverage comes in

• Problem:
  • When testing members of an equivalence class, an error of the associated data transformation may be masked on the path leading to this transformation

Error is masked on path c1→c2→c3
How path coverage comes in

• Problem:
- When testing members of an equivalence class, an error of the associated data transformation may be masked on the path leading to this transformation.

More about this in Session VI

Error is masked on path c1→c2→c3
Equivalence Classes, Pairwise Testing and Orthogonal Arrays

• **Application Situation.** Input vectors to SUT have so many components and / or so many possible values that the test of all parameter/value combinations is infeasible

• **Original application.** Combinatorial systems
Pairwise testing

• Recipe for pairwise testing with equivalence classes and orthogonal arrays

• Identify the **factors**: input and state parameters influencing SUT behavior

• Partition factor domains into **levels** (= equivalence classes)

• Use orthogonal arrays calculation technique to find input combinations such that

  • each parameter-level combination of given size $n$ occurs an equal number of times
Table 10: All Combinations for Three Variables of Three Levels Each

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Red</td>
<td>Red</td>
<td>Red</td>
</tr>
<tr>
<td>2</td>
<td>Red</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>3</td>
<td>Red</td>
<td>Red</td>
<td>Blue</td>
</tr>
<tr>
<td>4</td>
<td>Red</td>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td>5</td>
<td>Red</td>
<td>Green</td>
<td>Green</td>
</tr>
<tr>
<td>6</td>
<td>Red</td>
<td>Green</td>
<td>Blue</td>
</tr>
<tr>
<td>7</td>
<td>Red</td>
<td>Blue</td>
<td>Red</td>
</tr>
<tr>
<td>8</td>
<td>Red</td>
<td>Blue</td>
<td>Green</td>
</tr>
<tr>
<td>9</td>
<td>Red</td>
<td>Blue</td>
<td>Blue</td>
</tr>
<tr>
<td>10</td>
<td>Blue</td>
<td>Red</td>
<td>Red</td>
</tr>
<tr>
<td>11</td>
<td>Blue</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>12</td>
<td>Blue</td>
<td>Red</td>
<td>Blue</td>
</tr>
<tr>
<td>13</td>
<td>Blue</td>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td>14</td>
<td>Blue</td>
<td>Green</td>
<td>Green</td>
</tr>
<tr>
<td>15</td>
<td>Blue</td>
<td>Green</td>
<td>Blue</td>
</tr>
<tr>
<td>16</td>
<td>Blue</td>
<td>Blue</td>
<td>Red</td>
</tr>
<tr>
<td>17</td>
<td>Blue</td>
<td>Blue</td>
<td>Green</td>
</tr>
<tr>
<td>18</td>
<td>Blue</td>
<td>Blue</td>
<td>Blue</td>
</tr>
<tr>
<td>19</td>
<td>Green</td>
<td>Red</td>
<td>Red</td>
</tr>
<tr>
<td>20</td>
<td>Green</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>21</td>
<td>Green</td>
<td>Red</td>
<td>Blue</td>
</tr>
<tr>
<td>22</td>
<td>Green</td>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td>23</td>
<td>Green</td>
<td>Green</td>
<td>Green</td>
</tr>
<tr>
<td>24</td>
<td>Green</td>
<td>Green</td>
<td>Blue</td>
</tr>
<tr>
<td>25</td>
<td>Green</td>
<td>Blue</td>
<td>Red</td>
</tr>
<tr>
<td>26</td>
<td>Green</td>
<td>Blue</td>
<td>Green</td>
</tr>
<tr>
<td>27</td>
<td>Green</td>
<td>Blue</td>
<td>Blue</td>
</tr>
</tbody>
</table>

All-Combinations Testing

Pairwise testing with orthogonal arrays


<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Red</td>
<td>Red</td>
<td>Green</td>
</tr>
<tr>
<td>4</td>
<td>Red</td>
<td>Green</td>
<td>Red</td>
</tr>
<tr>
<td>9</td>
<td>Red</td>
<td>Blue</td>
<td>Blue</td>
</tr>
<tr>
<td>12</td>
<td>Blue</td>
<td>Red</td>
<td>Blue</td>
</tr>
<tr>
<td>14</td>
<td>Blue</td>
<td>Green</td>
<td>Green</td>
</tr>
<tr>
<td>16</td>
<td>Blue</td>
<td>Blue</td>
<td>Red</td>
</tr>
<tr>
<td>19</td>
<td>Green</td>
<td>Red</td>
<td>Red</td>
</tr>
<tr>
<td>24</td>
<td>Green</td>
<td>Green</td>
<td>Blue</td>
</tr>
<tr>
<td>26</td>
<td>Green</td>
<td>Blue</td>
<td>Green</td>
</tr>
</tbody>
</table>
Pairwise testing – Advantages

• Single-mode faults are detected
  • Faults that only depend on one parameter
  • Fault is revealed by selecting values of a certain class, regardless of the other parameters’ values

• Dual-mode faults are detected

• \( n \)-tupel value combinations are evenly distributed for \( n > 2 \)

• Some authors claim impressive test strength with surprisingly low number of test cases
Pairwise testing — Criticism

- Some experiments show that test strength is not as high as claimed by some others – perhaps due to “mechanical” application without analyzing the functional impact of each parameter?

- Some experiments show that same strength could be achieved (much easier) with random test data generation

- Only applicable to combinatorial systems

- Far more complicated – even inapplicable – if equivalence classes involve several parameters
How we apply pairwise testing in MBT

• **Objective.** Test “important” control state combinations in concurrent state machines

• **Strategy.**
  
  ● Select pairs based on writer-reader analysis
  
  ● Use orthogonal array methods so that control state distribution is “as even as possible”
  
  ● Use SMT solver to calculate the input traces needed to reach feasible control state combinations
Boundary Value Tests

• Boundary value testing refines equivalence class testing by selecting special representatives of each class who are at its boundary

• The intuitive meaning of a boundary value test $t$ is that a representative $t'$ of another equivalence class is “close” to $t$

• The formal meaning requires to look into metric spaces
• A **metric** on a space $X$ is a real-valued binary function $d$ fulfilling

\[ d : X \times X \to \mathbb{R} \]

\[ \forall x, y, z \in X : \]

\[ d(x, y) \geq 0 \text{ (non-negative)} \]

\[ d(x, y) = 0 \Rightarrow x = y \text{ (identity of indiscernibles)} \]

\[ d(x, y) = d(y, x) \text{ (symmetry)} \]

\[ d(x, z) \leq d(x, y) + d(y, z) \text{ (triangle inequality)} \]
At first, each atomic datatype is associated with a metric:

- Integral numbers, floating point numbers, enumerations (each enum interpreted by its integer value) and Booleans (true = 1, false = 0):

\[ d(x, y) = |x - y| \]

- Strings:

\[ d(x, y) = \text{Hamming-Distance}(x, y) \text{ or } d(x, y) = |\text{strcmp}(x, y)| \]
• The Hamming-Distance of two strings equals the number of character substitutions to be performed until they match each other.

• If \(x, y\) do not have the same length, the shorter one is padded with blanks; so it may always be assumed that the strings to be compared have equal length.

• The Hamming-Distance has the disadvantage that the places where the strings differ and the alphabetic distance of differing characters are not taken into account.

• As an alternative, `strcmp(3)` takes into account the distance between differing characters
• Based on the metric \( d(x,y) \) we introduce the concept of closeness for pairs \( x,y \) of values of an atomic type \( T \)

\[
\text{close}(x, y) \equiv d(x, y) > 0 \land (\forall z \in T - \{x\} : d(x, z) \geq d(x, y))
\]

• Observe that \( \text{close}(x,y) \) is also well-defined for floating point types, since for each \( x \) there is a “closest” \( y \) differing from \( x \) by one \( ulp \) (unit in the last place)
Automated **identification** of relevant test cases

Automated **generation** of concrete test data for test cases

— tool demonstration —
Automated execution of generated test procedures against System Under Test

Automated, documented tracing
Requirements → Test Cases → Test procedures → SUT functions

Automated generation of simulations and mutants:
check test suite strength

— tool demonstration —
Further Reading


