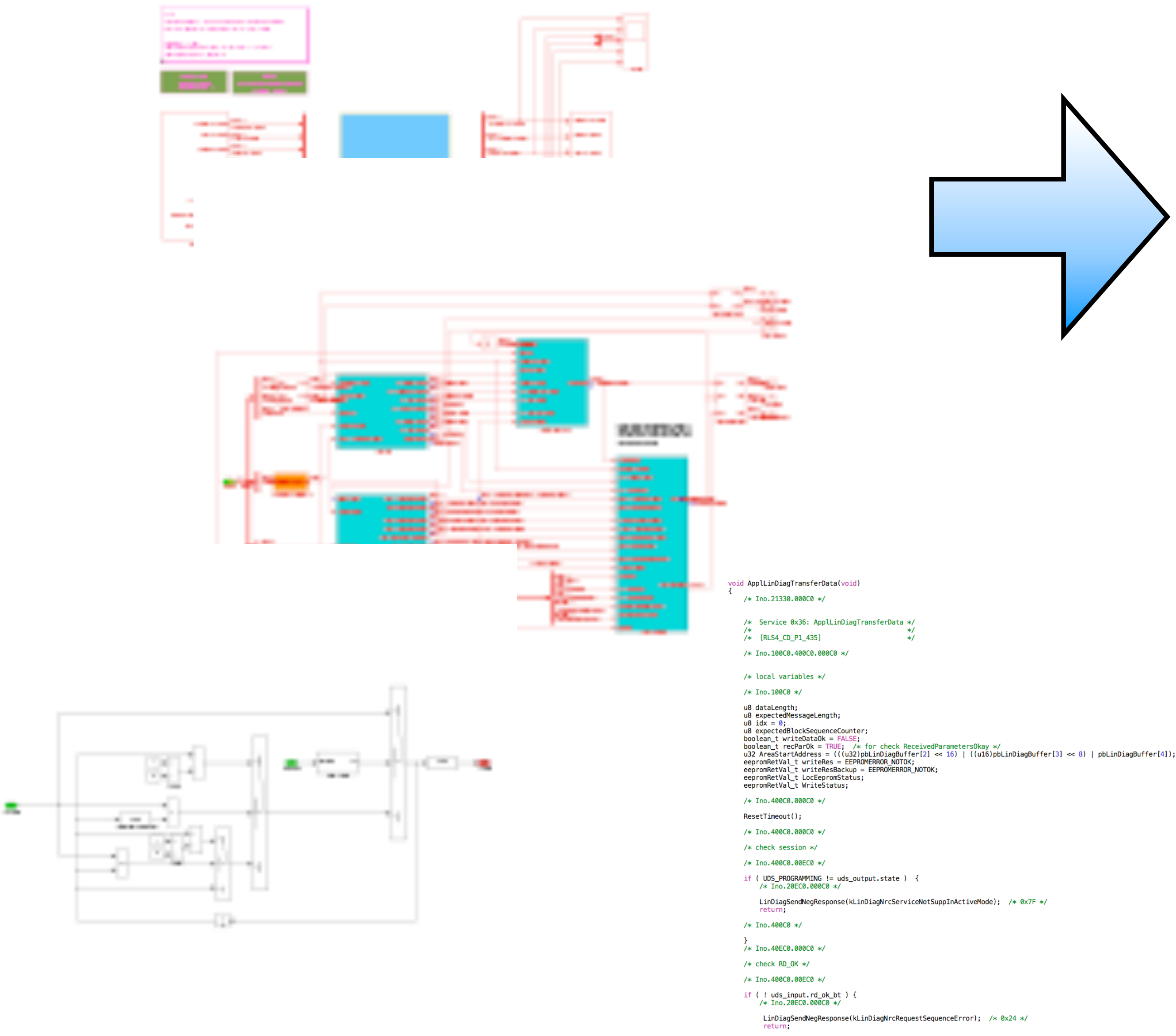
Testing Distributed Systems

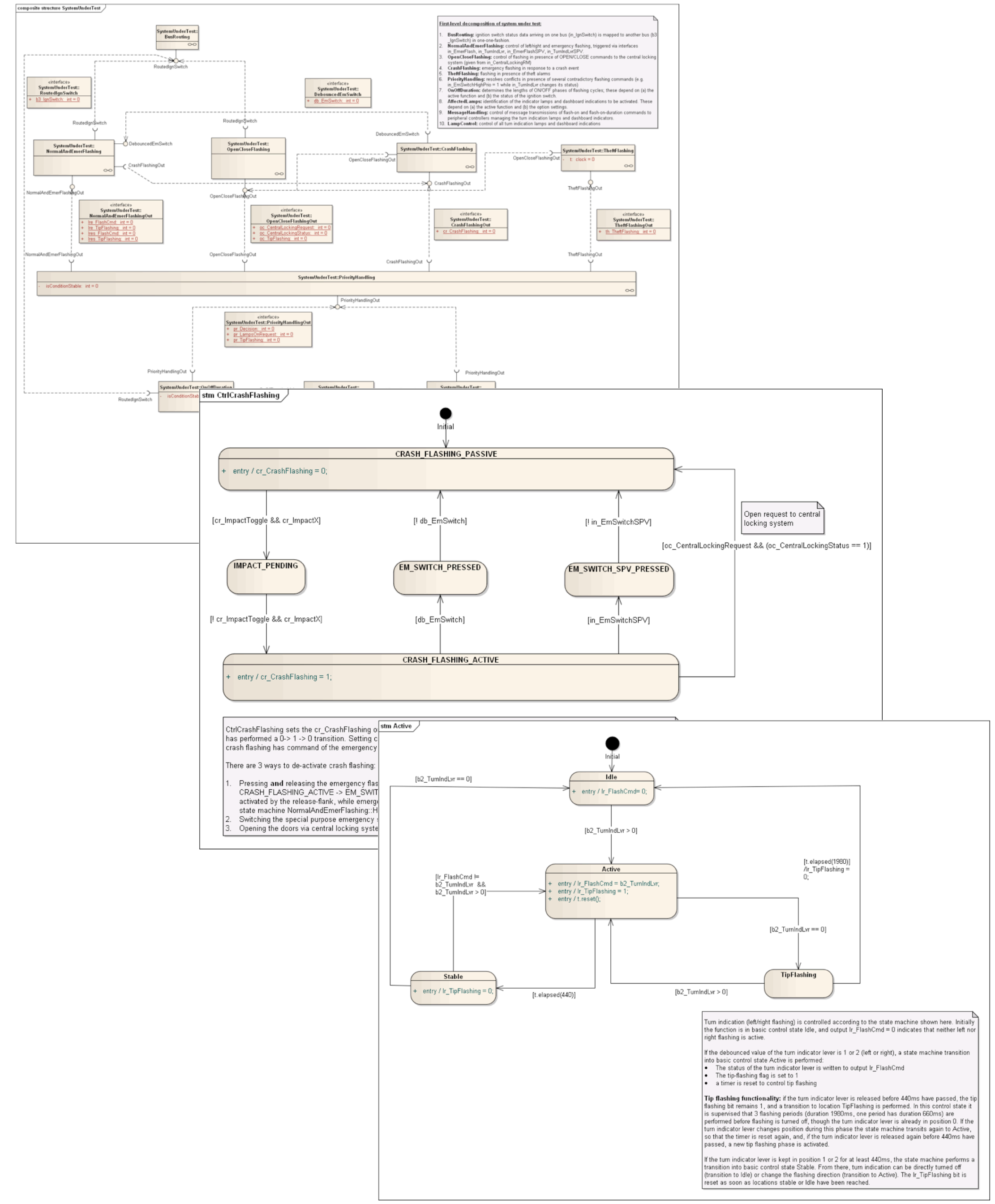
Part II: Test Cases, Model Coverage and Requirements Tracing — Coverage Measures for Distributed Systems
2012-08-01

Jan Peleska and Wen-ling Huang
University of Bremen
{jp,huang}@informatik.uni-bremen.de

Development Model (e.g. Simulink)

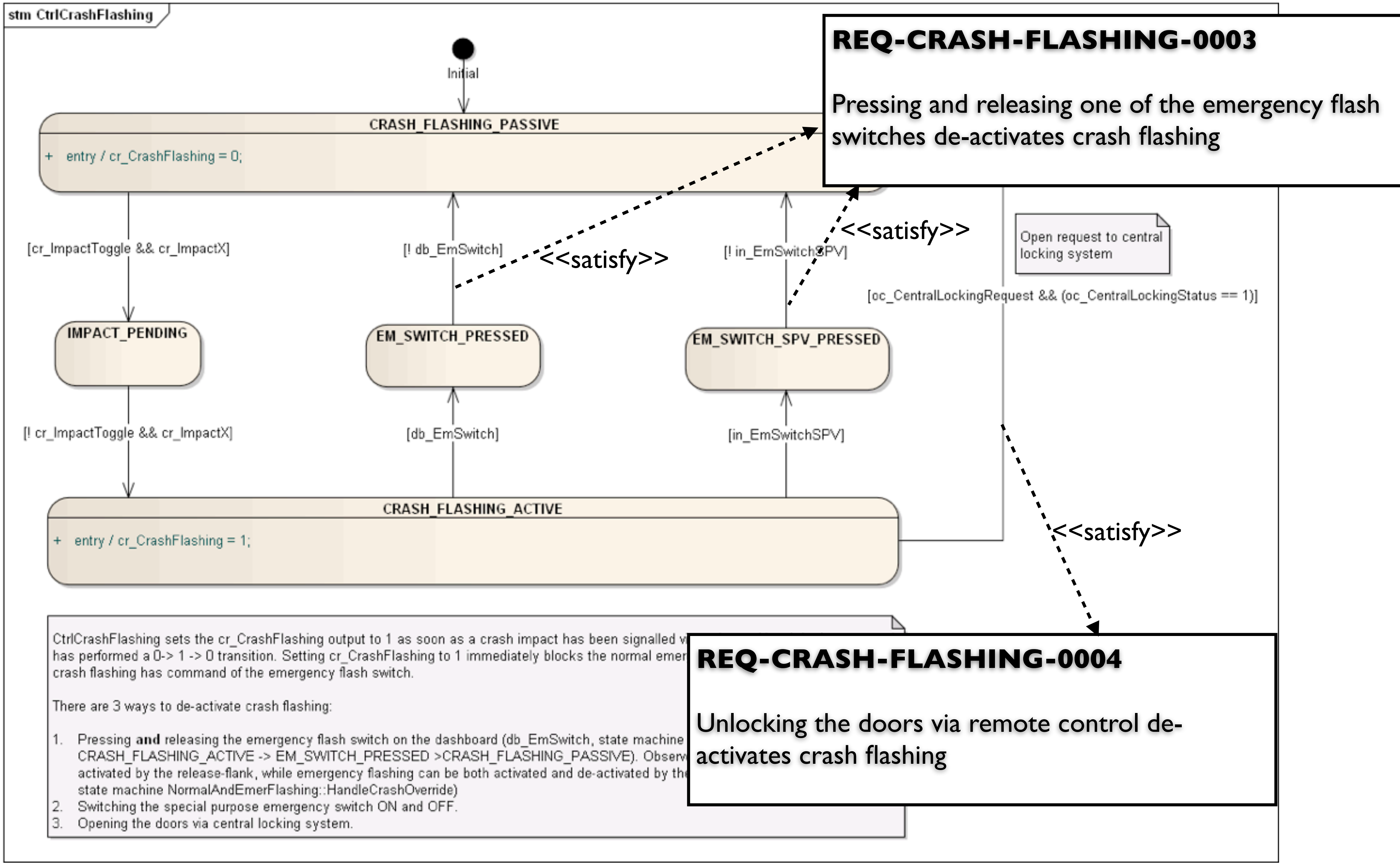


SysML Test Model

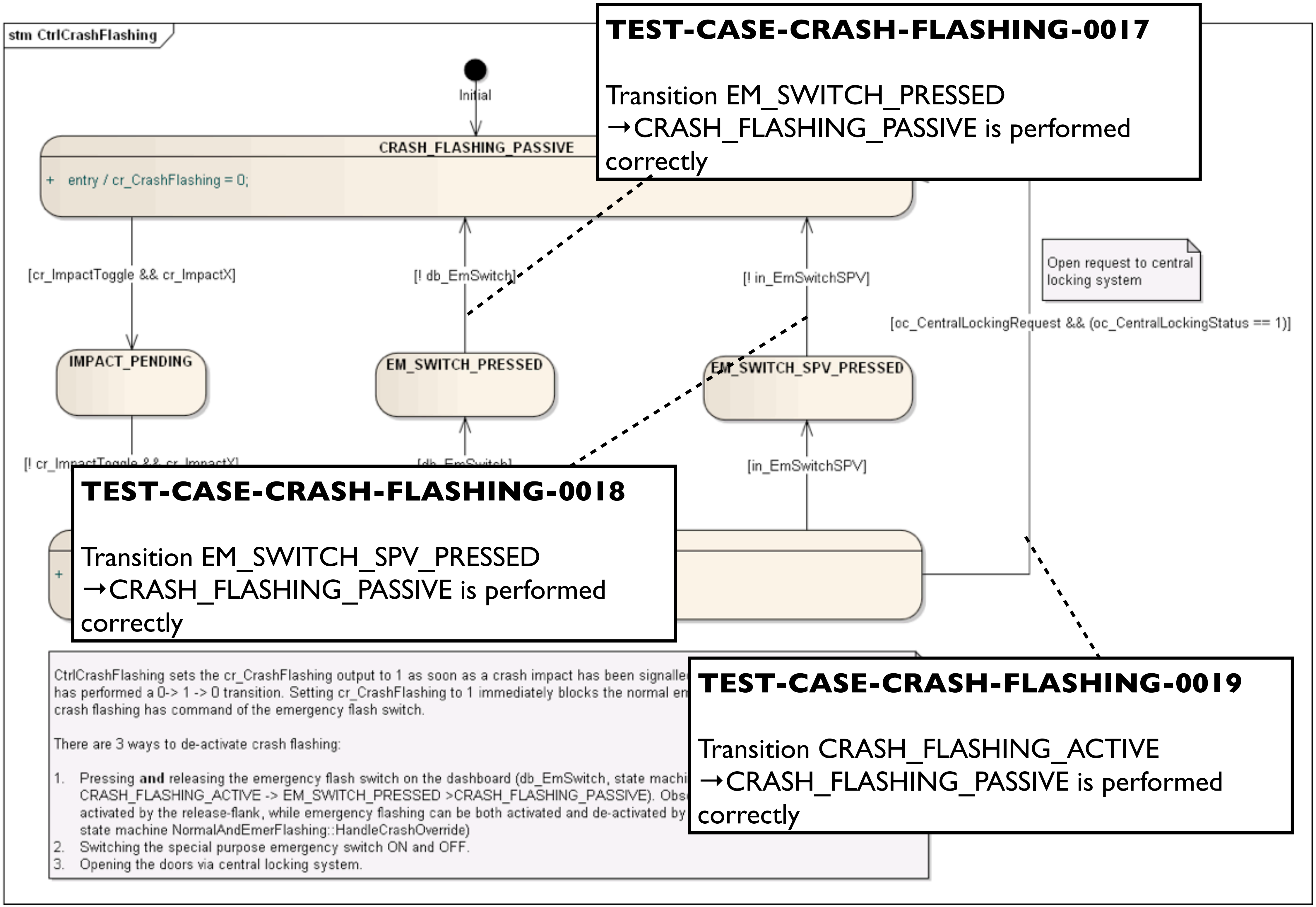


Manual derivation of test model from software requirements, developer model and hand-written code – performed during development model review

Requirements tracing to model elements – SysML method



Automated generation of model coverage test cases



Automated tracing from test cases to requirements

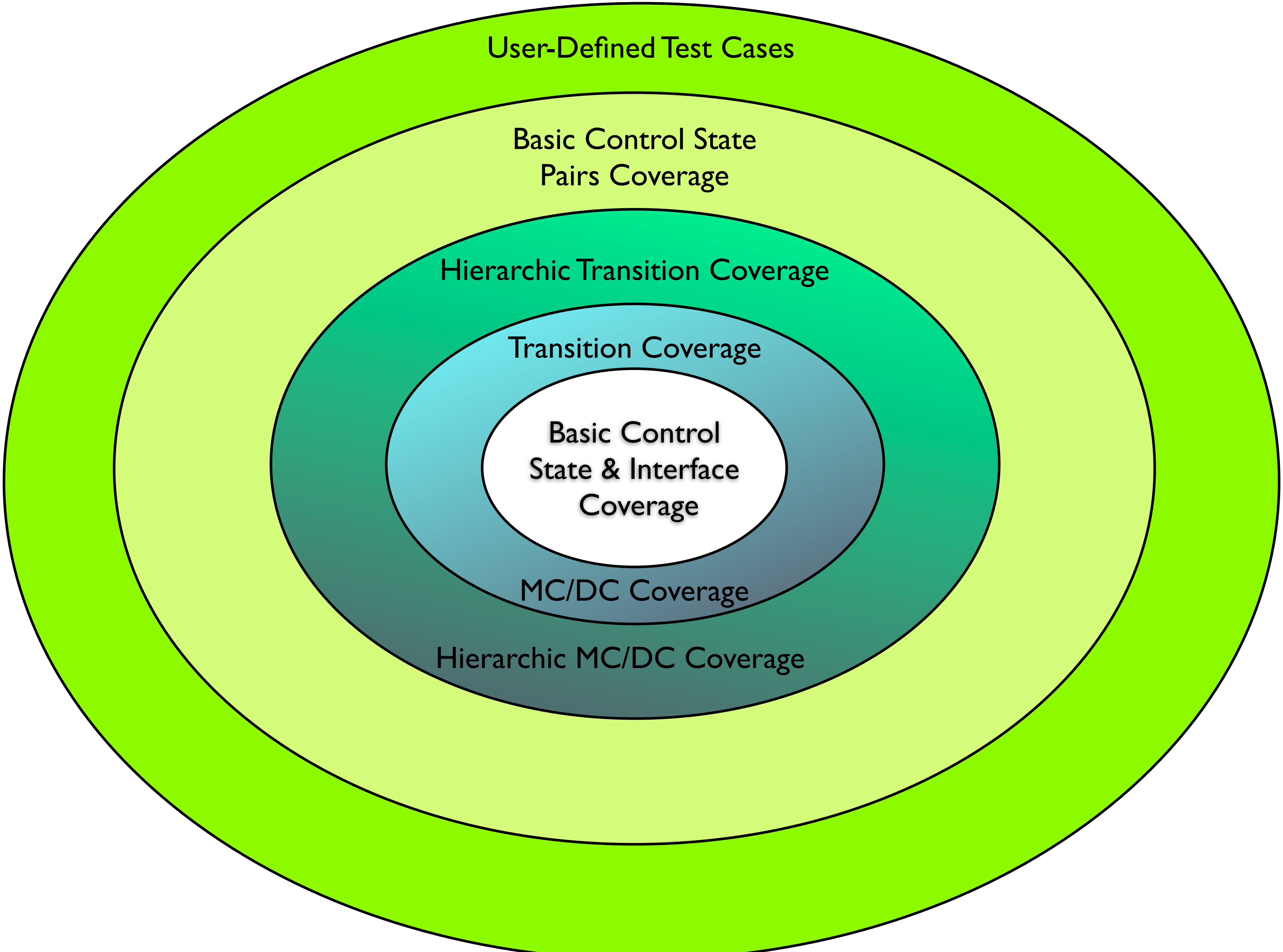
Requirement	Tested by
REQ-CRASH-FLASHING-0003	TEST-CASE-CRASH-FLASHING-0017
	TEST-CASE-CRASH-FLASHING-0018
REQ-CRASH-FLASHING-0004	TEST-CASE-CRASH-FLASHING-0019
...	...

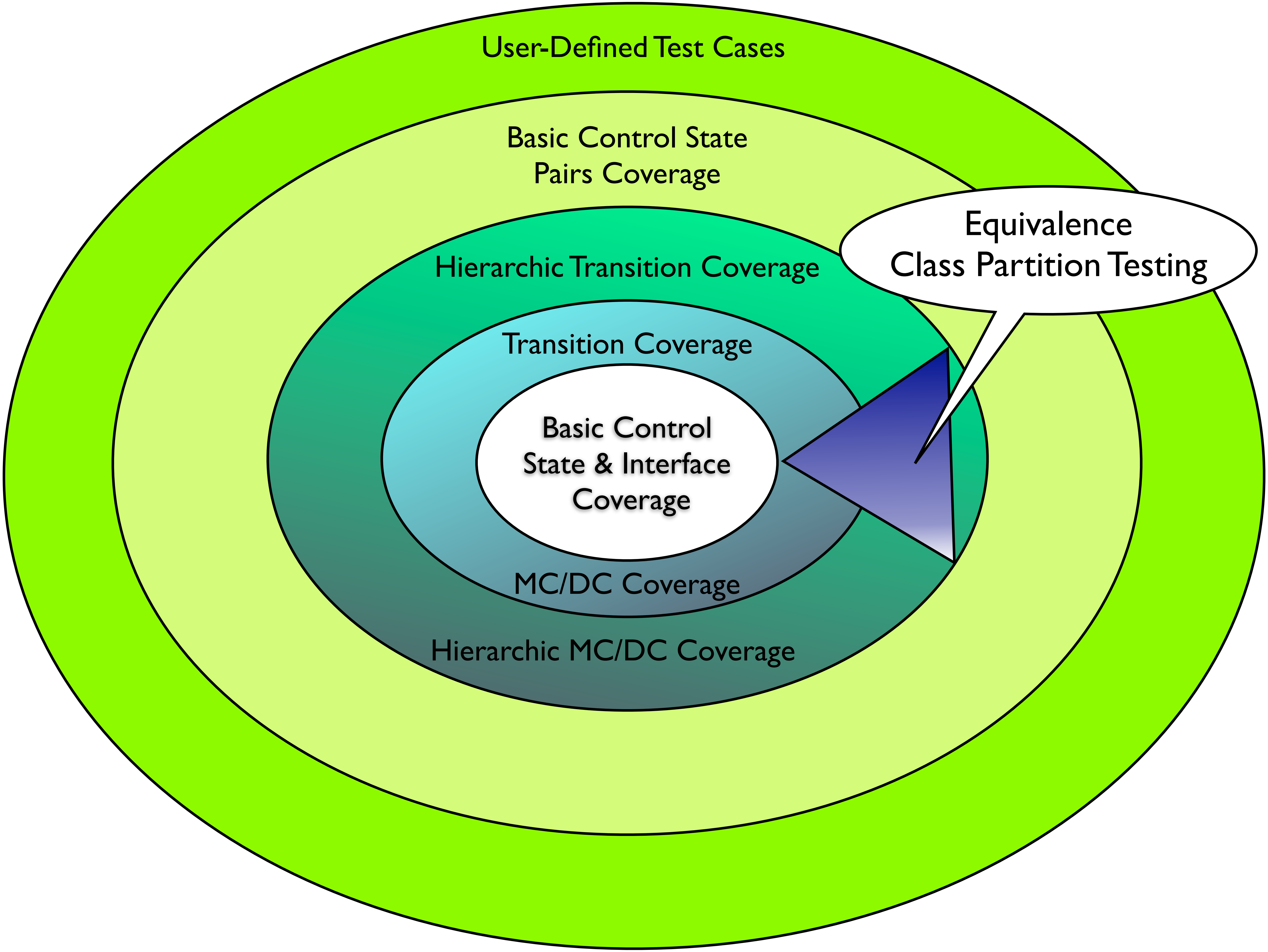
Automated tracing from test cases to development model components

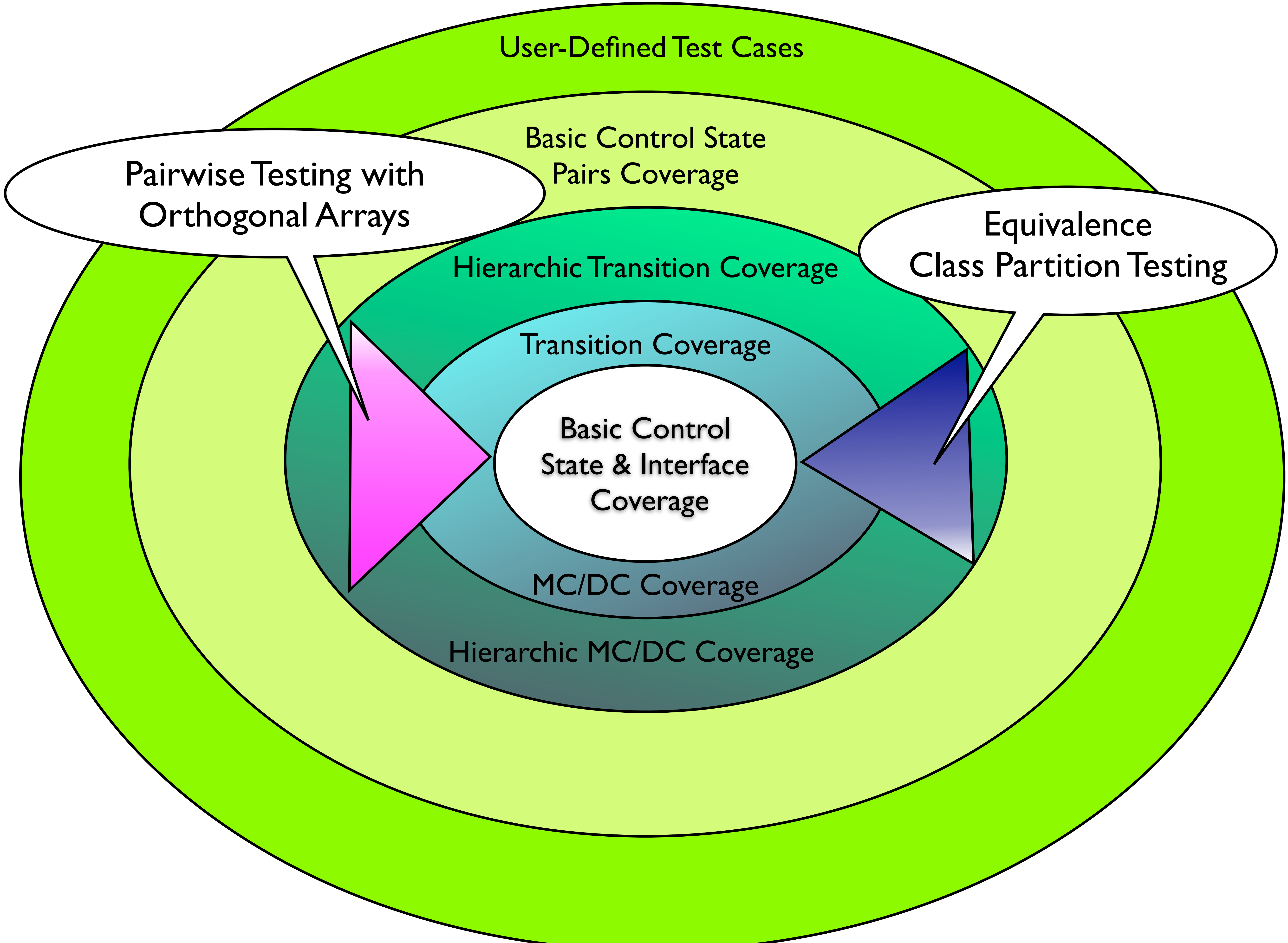
Simulink Component	Tested by
Simulink.Crash_flashing.component_x	TEST-CASE-CRASH-FLASHING-0017 TEST-CASE-CRASH-FLASHING-0018 TEST-CASE-CRASH-FLASHING-0019
...	...

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
Pairs Coverage
Hierarchic Transition Coverage
Transition Coverage
Basic Control
State & Interface
Coverage
MC/DC Coverage
Hierarchic MC/DC Coverage


User-Defined Test Cases
Basic Control State
Pairs Coverage
Hierarchic Transition Coverage
Equivalence
Class Partition Testing
Transition Coverage
Basic Control
State & Interface
Coverage
MC/DC Coverage
Hierarchic MC/DC Coverage


User-Defined Test Cases
Basic Control State
Pairs Coverage
Pairwise Testing with
Orthogonal Arrays
Equivalence
Class Partition Testing
Hierarchic Transition Coverage
Transition Coverage
Basic Control
State & Interface
Coverage
MC/DC Coverage
Hierarchic MC/DC Coverage

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

  If 2 elements xo and x1 are members of the same partition (= equivalence class), it may be expected that every error uncovered by x0 will also be uncovered by x1

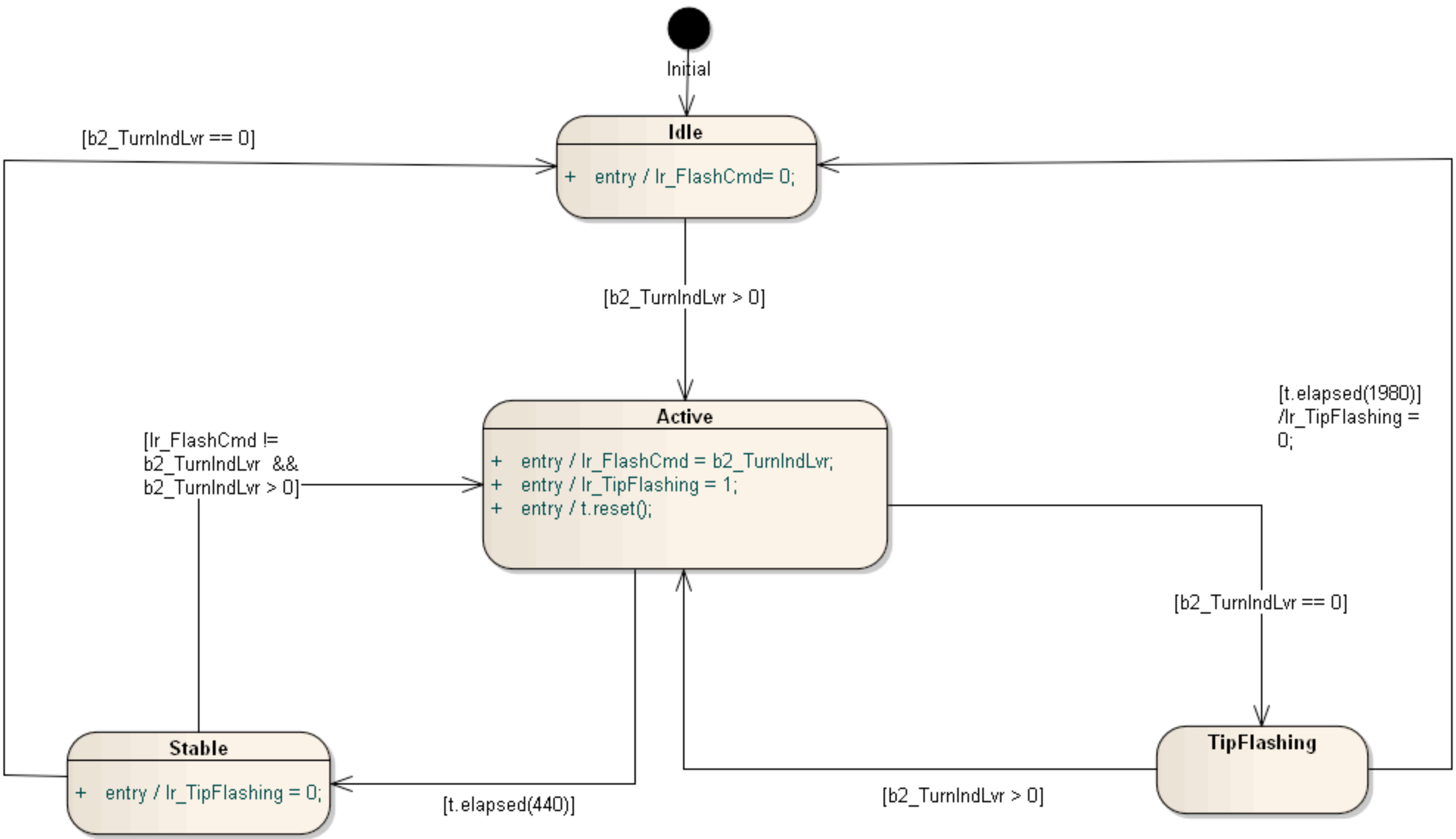
Equivalence Class Partition Testing

- **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

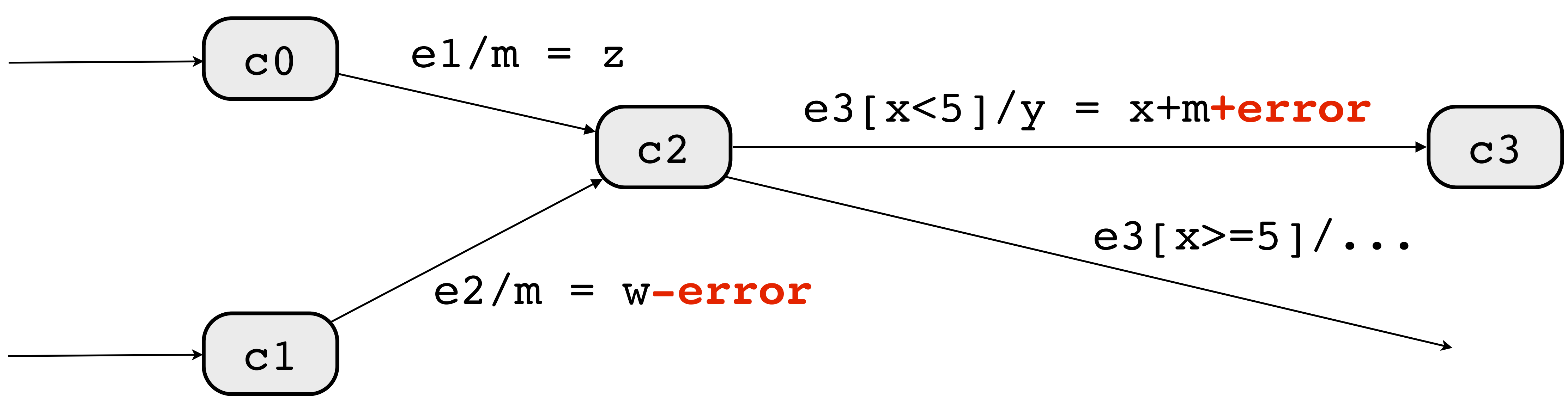


3 Types of Equivalence Classes

- **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

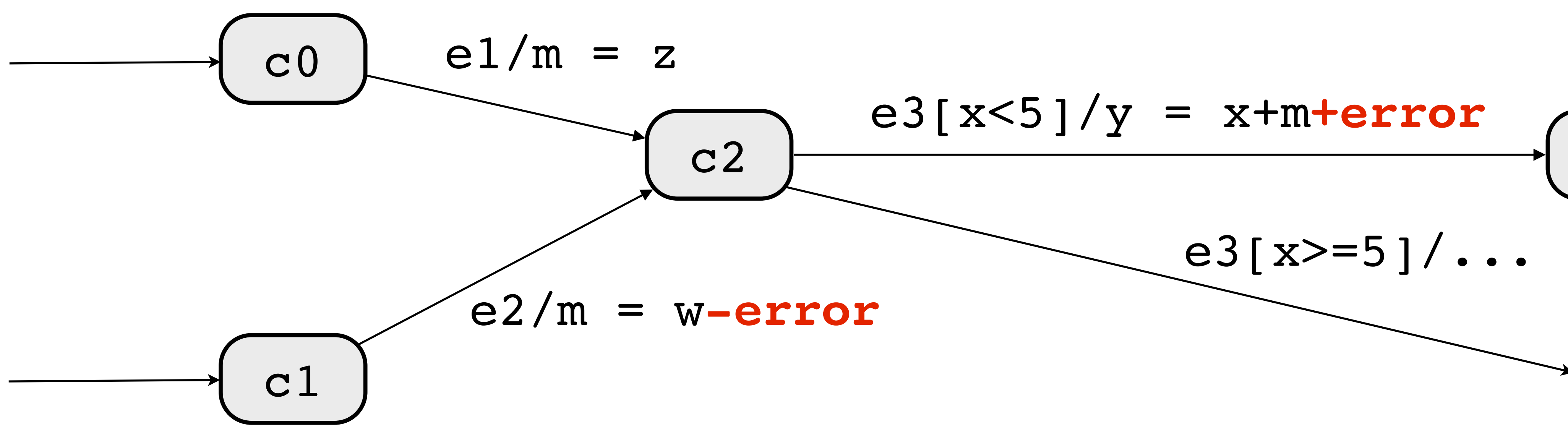


Error is masked on path c1→c2→c3

How path coverage comes in

More about this in Session VI

- 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

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			
	A	B	C
1	Red	Red	Red
2	Red	Red	Green
3	Red	Red	Blue
4	Red	Green	Red
5	Red	Green	Green
6	Red	Green	Blue
7	Red	Blue	Red
8	Red	Blue	Green
9	Red	Blue	Blue
10	Blue	Red	Red
11	Blue	Red	Green
12	Blue	Red	Blue
13	Blue	Green	Red
14	Blue	Green	Green
15	Blue	Green	Blue
16	Blue	Blue	Red
17	Blue	Blue	Green
18	Blue	Blue	Blue
19	Green	Red	Red
20	Green	Red	Green
21	Green	Red	Blue
22	Green	Green	Red
23	Green	Green	Green
24	Green	Green	Blue
25	Green	Blue	Red
26	Green	Blue	Green
27	Green	Blue	Blue

All-Combinations Testing

Example from http://www.developsense.com/pairwiseTesting.html

Table 11: All-Pairs Array, Three Variables of Three Levels Each			
	A	B	C
2	Red	Red	Green
4	Red	Green	Red
9	Red	Blue	Blue
12	Blue	Red	Blue
14	Blue	Green	Green
16	Blue	Blue	Red
19	Green	Red	Red
24	Green	Green	Blue
26	Green	Blue	Green

Pairwise testing with orthogonal arrays

Example from http://www.developsense.com/pairwiseTesting.html

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

$$
\begin{aligned}
&d : X \times X \to \mathbf{R} \\
&\forall x, y, z \in X : \\
&\quad d(x,y) \geq 0 \quad \text{(non-negative)} \\
&\quad d(x,y) = 0 \Rightarrow x = y \quad \text{(identity of indiscernibles)} \\
&\quad d(x,y) = d(y,x) \quad \text{(symmetry)} \\
&\quad d(x,z) \leq d(x,y) + d(y,z) \quad \text{(triangle inequality)}
\end{aligned}
$$

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:

$$d(x, y) = \text{Hamming-Distance}(x, y) \quad \text{or} \quad d(x, y) = |\text{strcmp}(x, y)|$$

- 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*

$$\mathrm{close}(x,y) \equiv d(x,y) > 0 \wedge (\forall z \in T - \{x\} : d(x,z) \geq d(x,y))$$

- Observe that *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

1. Taguchi, G. 1987. System of Experimental Design, Volume 1 & 2. UNIPUB/Krass International Publications.

2. Tatsumi, K. 1987. Test case design support system. In International Conference on Quality Control (ICQC), Tokyo. 615–620.

3. Phadke, M. S. 1989. Quality Engineering Using Robust Design. Prentice Hall, Englewood Cliff, NJ.

4. Bach, J. and Schroeder, P. 2004. Pairwise testing - a best practice that isnt. In 22nd Pacific Northwest Software Quality Conference. 180–196.

5. Huang, W.-l. and Peleska, J. 2012. Specialised Test Strategies. Technical Note Number: D34.2 Version: 0.1 Date: June 2012, Public Document. http://www.compass-research.eu