Fehlerbasiertes Testen

Alexander Pretschner, TU München
Karlsruhe Developer Days, 21/5/2014

Agenda

- Good tests
- Partition-based testing
- Why coverage shouldn‘t be used a-priori
- Fault models
- Testing based on fault models
- Discussion

Agenda

- Good tests
- Partition-based testing
- Why coverage shouldn‘t be used a-priori
- Fault models
- Testing based on fault models
- Discussion

What's a good test case?

- "Ability to detect failures"
  - No good test cases for a perfect program!
- "Ability to detect potential failures"
  - "Potential"? Effort?
- "Ability to detect potential (or: likely) failures with good cost-effectiveness"
    - Writing/executing/evaluating/maintaining the test
    - Remaining failures in the field—severity
    - Going from failure to fault
- Perfect! And useless!

Coverage-Based Testing

- Challenge: operational, measurable quality of tests
  - „Adequacy“: selection, stopping, assessment criteria
- Coverage one response

- … a good response?

Agenda

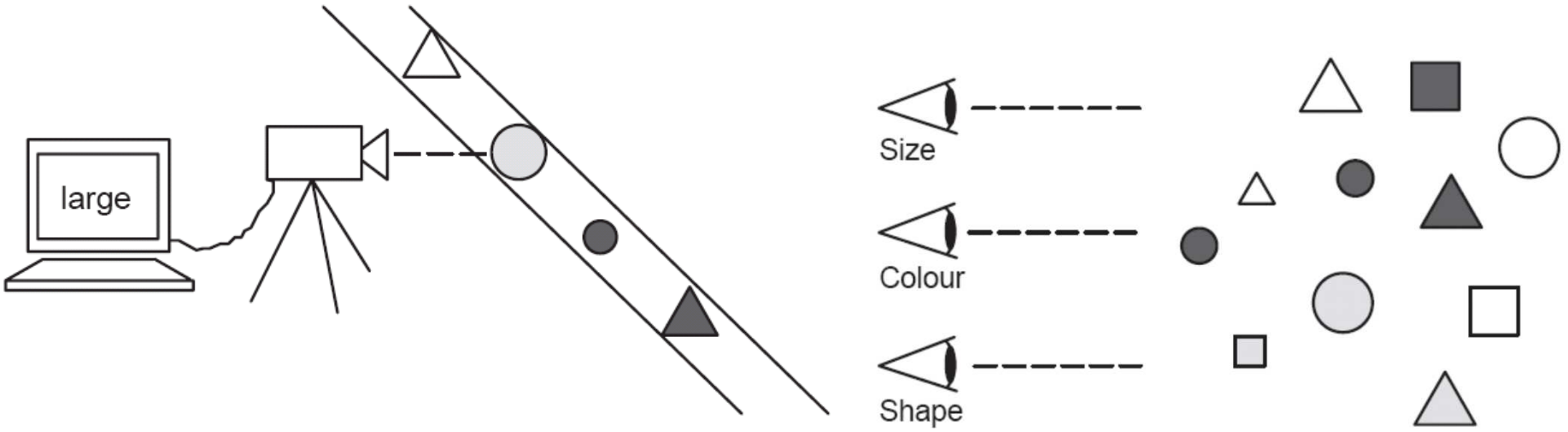
- Good tests?
- Partition-based testing
- Why coverage shouldn‘t be used a-priori
- Fault models
- Testing based on fault models
- Discussion

Partition-Based Testing

- Adequacy criteria induce partition of input domain
  - Requirements
  - Coverage criteria
  - [Faults]

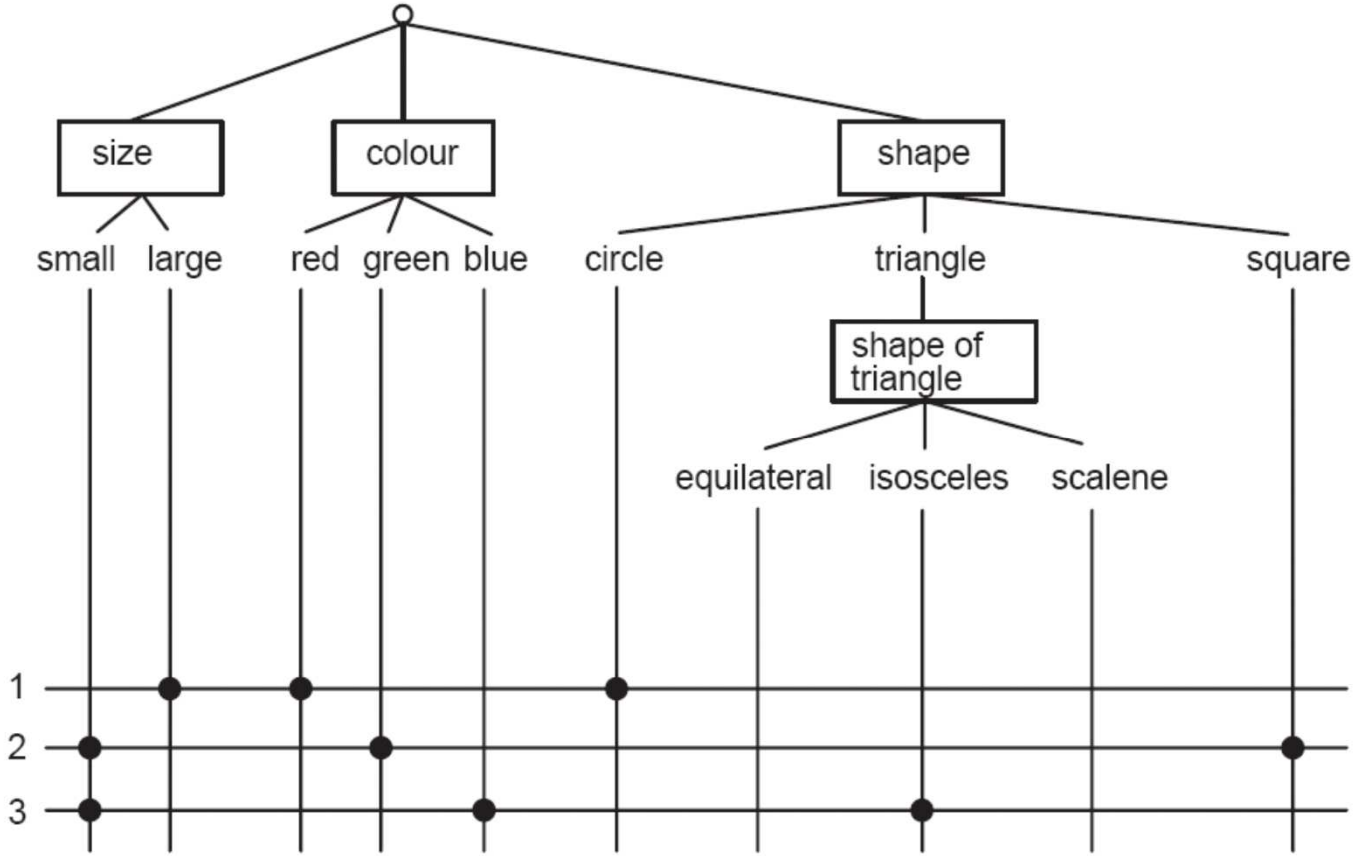
Input space partition: category-partition method

- Consider input space “under various aspects”
- For each “aspect”, form disjoint and complete set of classes
- (Iterate: build recursive classification)
- Instantiate classes so that the input domain is “covered”



[Grochtmann, Wegener, Grimm ’95]

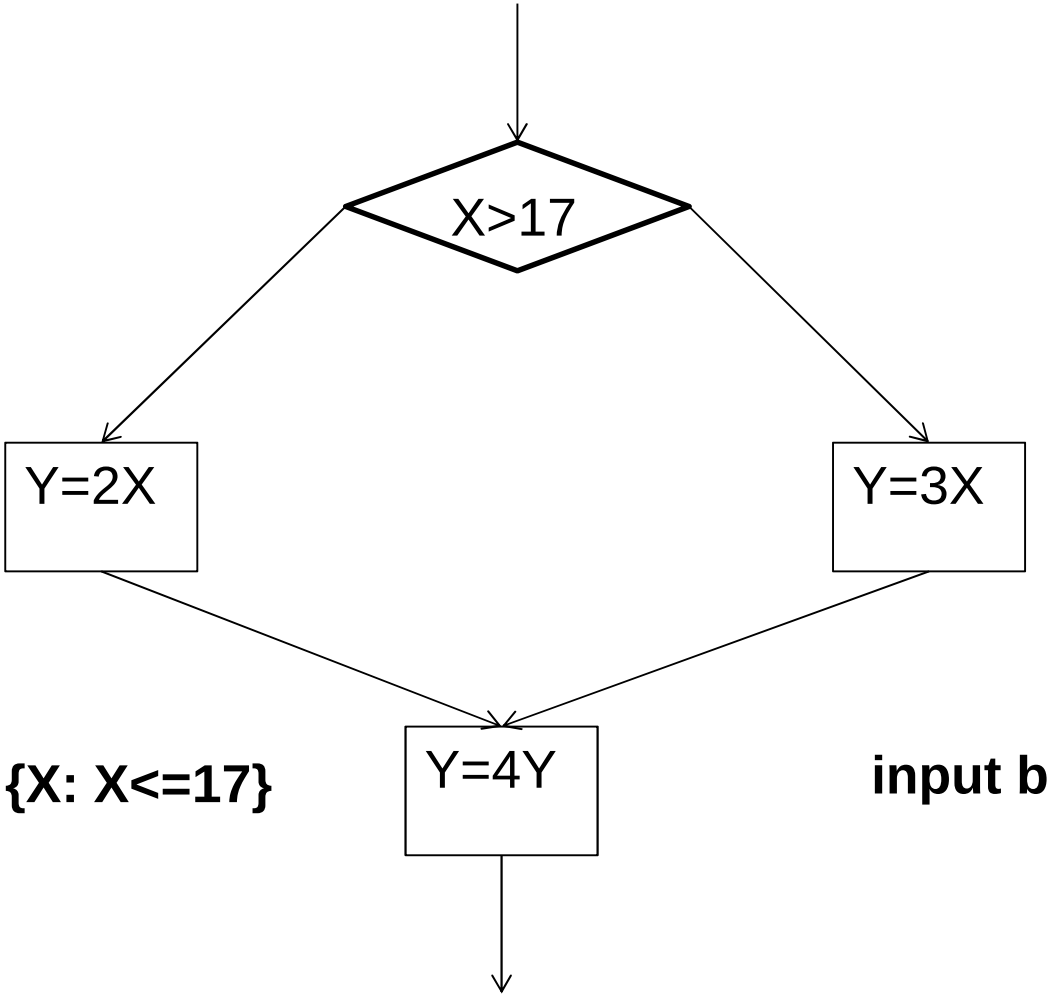
Input space partition: category-partition method



[Grochtmann, Wegener, Grimm ’95]

Input Space Partitioning: Coverage Criteria

```
if(X>17)
  Y=2*X;
else
  Y=3*X;
Y=4*Y;
```



input block 1: {X: X<=17}

input block 2: {X: X>17}

Bottom line

- Coverage-based testing instance of partition-based testing

Agenda

- Good tests?
- Partition-based testing
- Why coverage shouldn‘t be used a-priori
- Fault models
- Testing based on fault models
- Discussion

Simple decision

Pick two test cases for

```
if x==1
   f(g(h(i(j(k(l(m(x))))))))
else
   m(l(k(j(i(h(g(f(x))))))))
endif
```

[nondeterministic f..m]

Simple decision

Now, pick two test cases for

```
if x==1
   f(g(h(i(j(k(l(m(x)))))))
else
   f(g(h(x)))
endif
```

Simpler decision

And now, pick two test cases for

```
if x==1
   f(g(h(i(j(k(l(m(x))))))))
else
   print „es war sehr schön, es hat mich sehr gefreut“
endif
```

So what?

- Structural criterion a good idea?

- Fault model matters!

Disclaimer

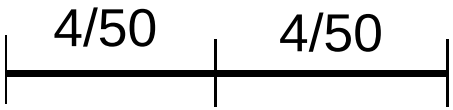
- In this case, the truth is somewhat more complicated: coverage criteria usually applied to all function definitions, not just the main function
- General idea applicable nonetheless
- Plenty of empirical evidence that coverage is not helpful when used a-priori, mixed findings for a-posteriori usage
  most recent [Inozemtseva&Reid‘14]

Failure Detection Abilities

- P_r and P_p for comparing random and partition testing in terms of failure detection
  - Overall failure rate θ: probability that a failure-causing input will be selected as test case
  - $\theta_i = m_i/|D_i|$: the failure rate of subdomain D_i ; m_i number of failure causing inputs in D_i
- Probability that random testing causes at least one failure is
  $P_r = 1-(1-\theta)^n$
- Probability that partition testing causes at least one failure is
  $P_p = 1 - \prod_{1 \le i \le k}(1- \theta_i)^{ni}$
  - $1 \le n_i \le |D_i|$ number of tests randomly chosen from subdomain D_i
- Assume $\sum_{1 \le i \le k} n_i = n$—same number of tests for both methods

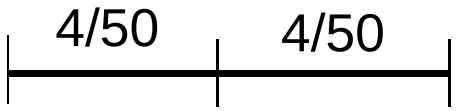
Random and Partition Testing

- Partition testing can be better, worse, or the same as random testing
  - d=100, 8 inputs failure-causing, n=2 tests to be selected
  - $P_r=1-(1-.08)^2=.15$
- k=2 subdomains
  - $P_p=1-(1-4/50)^2$ = P_r

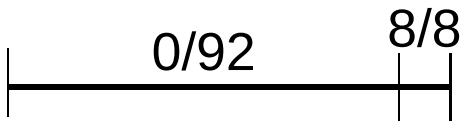


Random and Partition Testing

- Partition testing can be better, worse, or the same as random testing
  - d=100, 8 inputs failure-causing, n=2 tests to be selected
  - $P_r=1-(1-.08)^2=.15$
- k=2 subdomains
  - $P_p=1-(1-4/50)^2 = P_r$

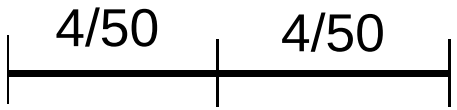


  - $P_p=1 > P_r$

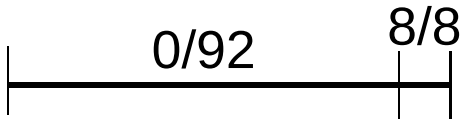


Random and Partition Testing

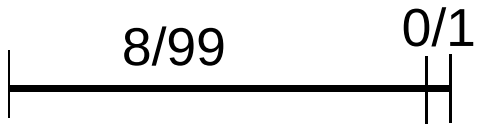
- Partition testing can be better, worse, or the same as random testing
  - d=100, 8 inputs failure-causing, n=2 tests to be selected
  - $P_r=1-(1-.08)^2=.15$
- k=2 subdomains
  - $P_p=1-(1-4/50)^2 = P_r$



  - $P_p=1 > P_r$



  - $P_p=1-(1-0/1)*(1-8/99)=.08 < P_r$



Results (Weyuker&Jeng 1991; Gutjahr 1999)

- In general, partition based can be as good as, better than, or worse than random testing
  - **Fault-prone partitions not known in advance**
- [yes several reasonable objections to this model]
- [Generalization to θ_i modeled as random variables: **If $E(\theta_i)$ the same for all blocks** $1 \leq i \leq k$ same number of tests from each block, **then $E(P_r) \leq E(P_p)$**]

Discussion

- If a-priori failure likelihoods are not known (or their characteristics or characteristics of their expectation), then partition-based testing **can be good or bad**!
- Yes, coverage is good from a management perspective.
  Yes, MC/DC coverage is required by DO 178-B.
  Yes, we can automate the derivation of tests.

- **But, we do it because we can and because one number is better than no number, not because it would, from a failure detection perspective, make sense!**

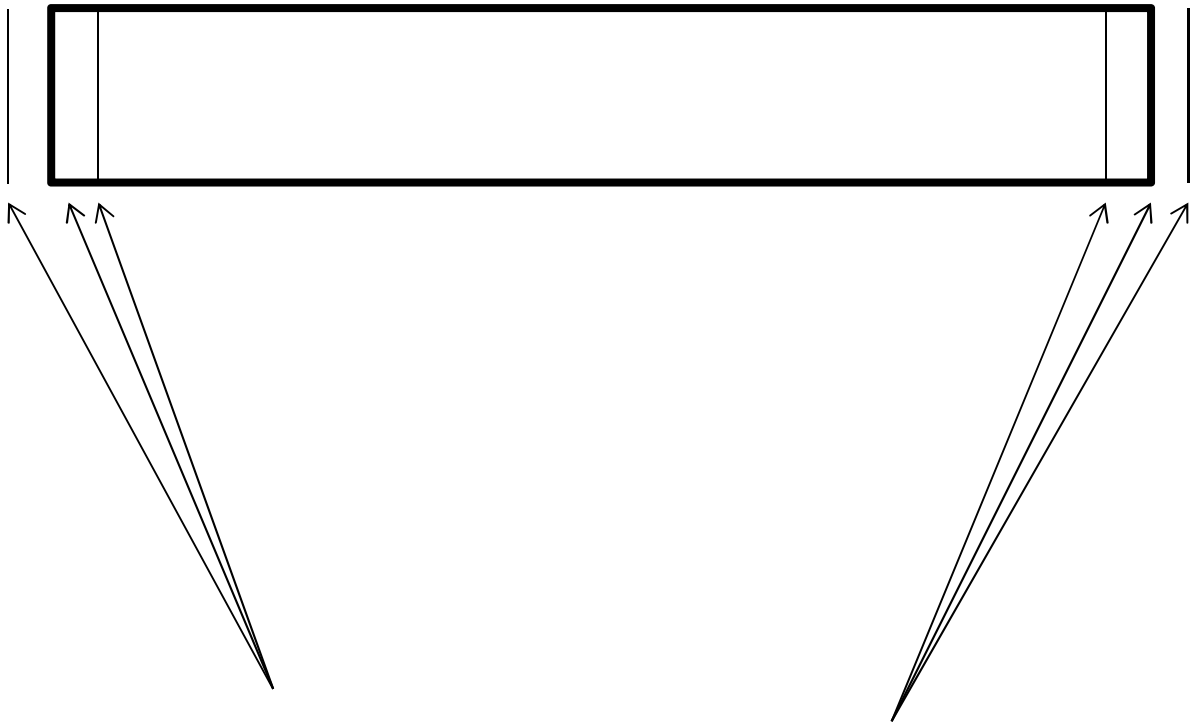
Disclaimer II

- Random testing really such a good idea?

Agenda

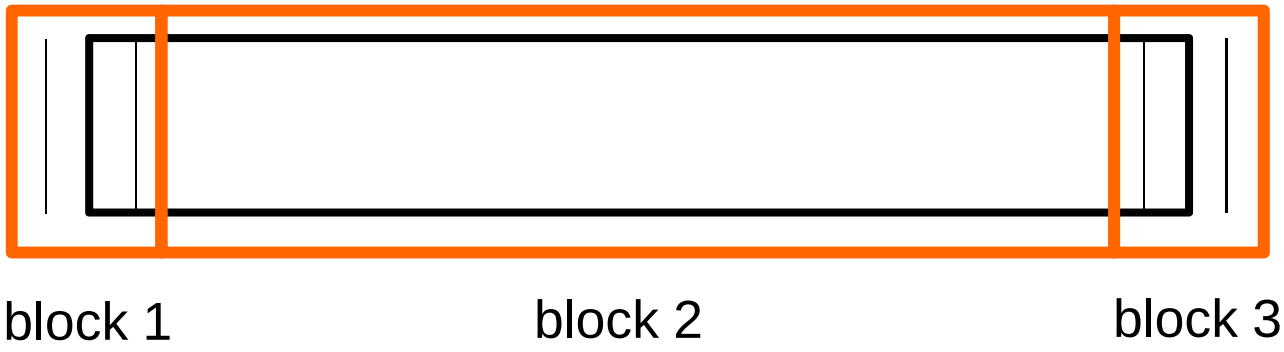
- Good tests?
- Partition-based testing
- Why coverage shouldn‘t be used a-priori
- Fault models
- Testing based on fault models
- Discussion

Limit testing?



Empirically speaking, trouble!

Limit testing?



Blocks 1 and 3 with higher expected failure rates
Plus, comparably small w.r.t. block 2
Hence: can expect E(Pp)>E(Pr)

What‘s this?

What‘s this?

- … a fault model!

Fault models

- Limit testing
- Deadlocks, order violations, atomicity violations
- Incorrect transition, sneak paths, trap doors, corrupt states
  …
- Invariant violations in subclass
- Syntactic problems as used in mutation testing
- Combinatorial testing

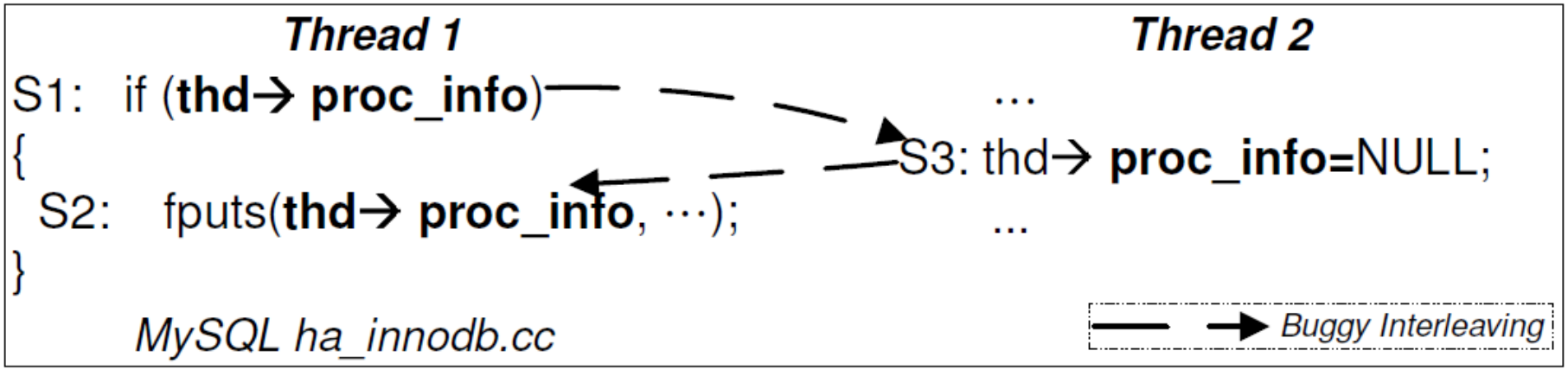
- Domain-specific faults

Fault models

- Limit testing
- Deadlocks, order violations, atomicity violations
- Incorrect transition, sneak paths, trap doors, corrupt states …
- Invariant violations in subclass
- Syntactic problems as used in mutation testing
- Combinatorial testing

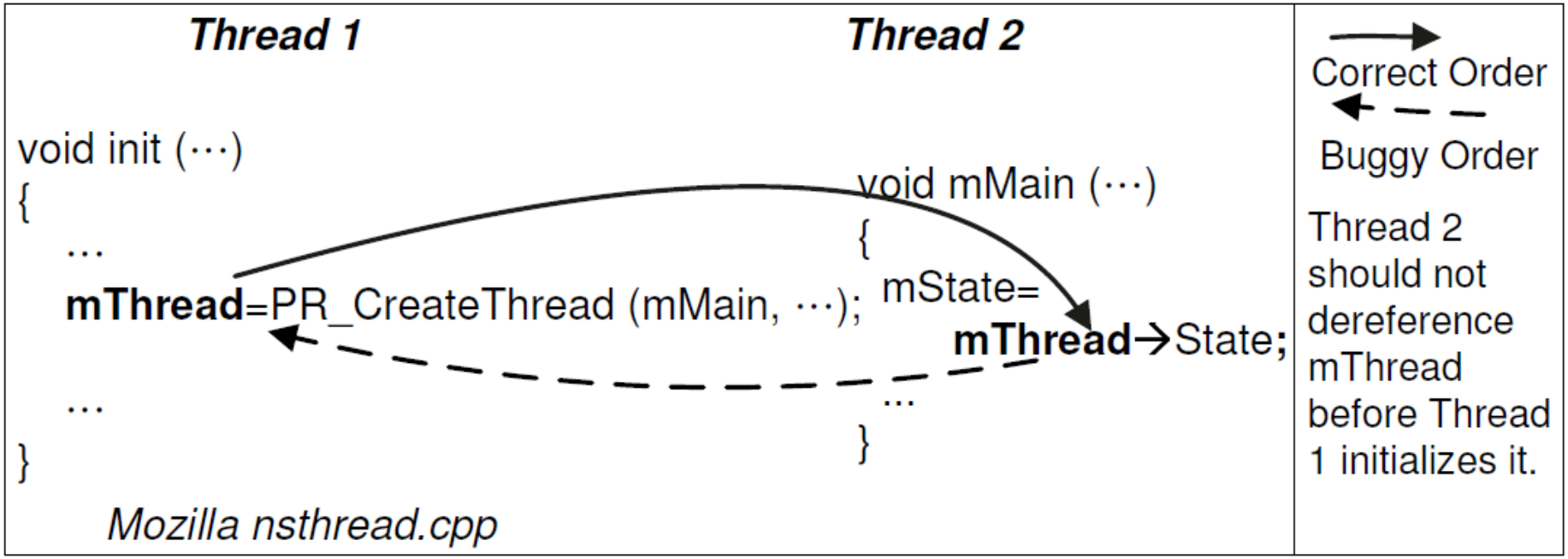
- Domain-specific faults

Non-Deadlock: Atomicity Violations



Shan Lu, Soyeon Park, Eunsoo Seo and Yuanyuan Zhou : Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics. Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, pp. 329-339, 2008

Non-Deadlock: Order Violation



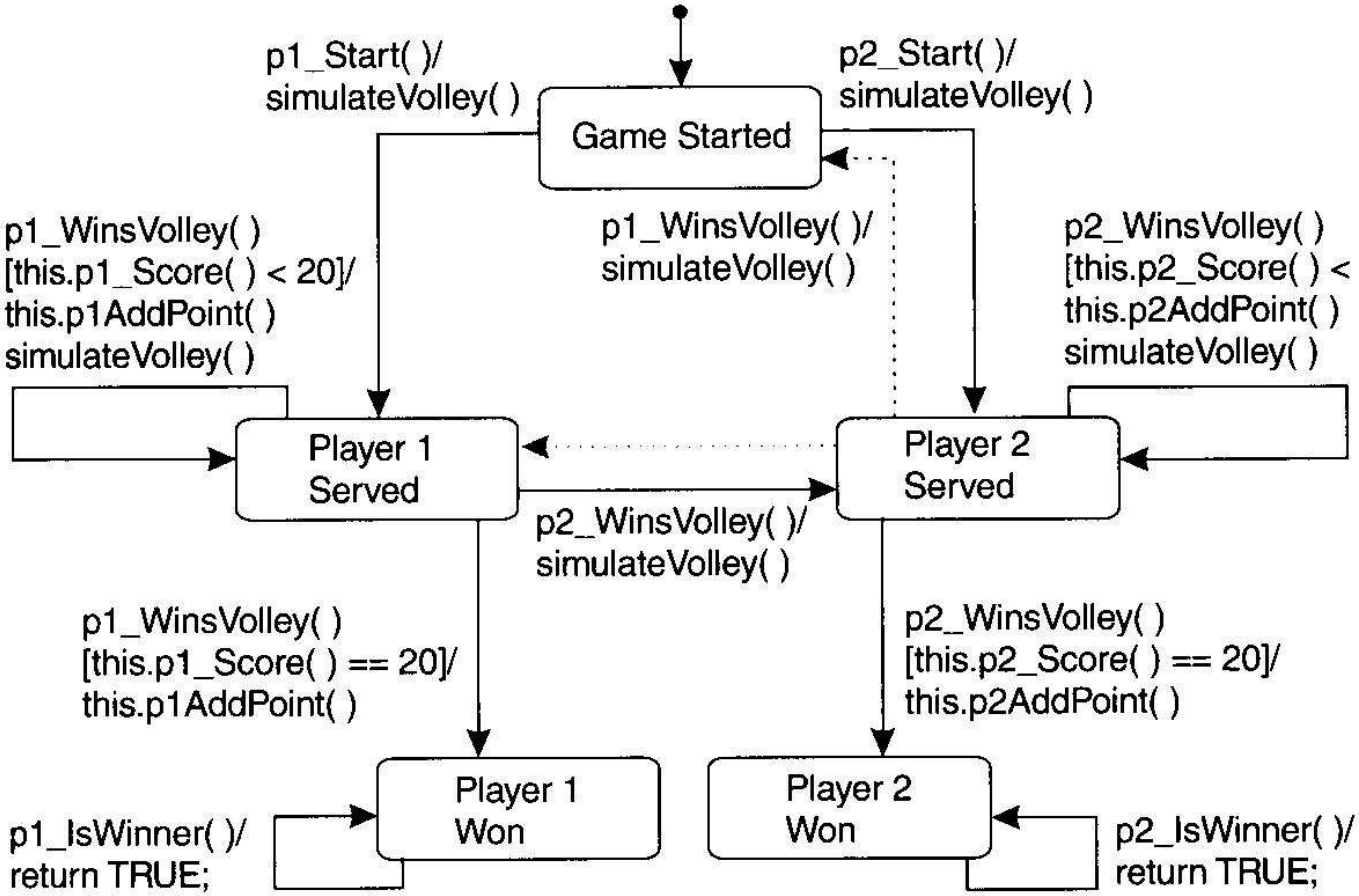
Shan Lu, Soyeon Park, Eunsoo Seo and Yuanyuan Zhou : Learning from Mistakes – A Comprehensive Study on Real World Concurrency Bug Characteristics. Proceedings of the 13th international conference on Architectural support for programming languages and operating systems, pp. 329-339, 2008

Fault models

- Limit testing
- Deadlocks, order violations, atomicity violations
- Incorrect transition, sneak paths, trap doors, corrupt states …
- Invariant violations in subclass
- Syntactic problems as used in mutation testing
- Combinatorial testing

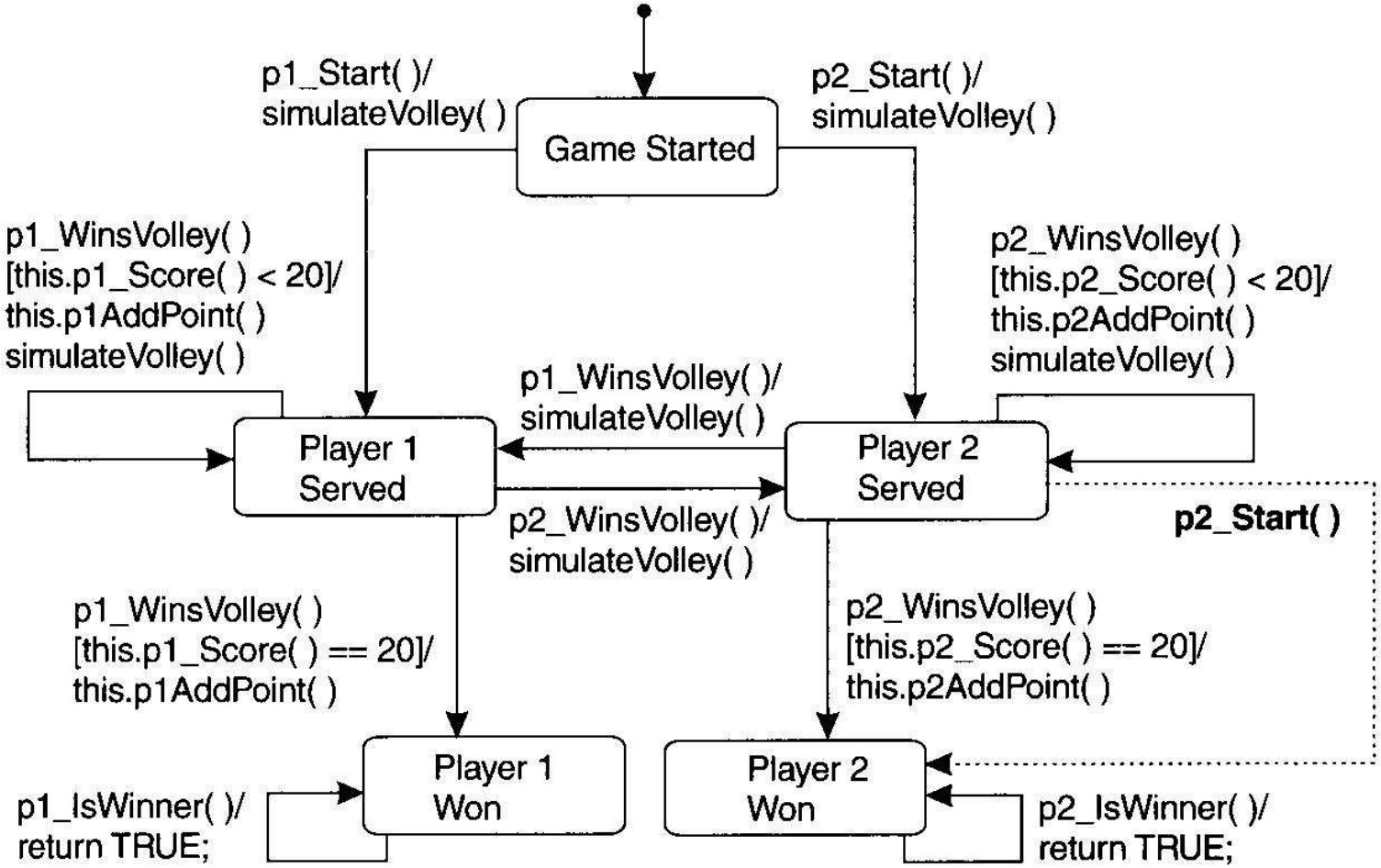
- Domain-specific faults

Incorrect Transition



Binder, Testing OO Systems, 1999

Sneak Path



Binder, Testing OO Systems, 1999

Fault models

- Limit testing
- Deadlocks, order violations, atomicity violations
- Incorrect transition, sneak paths, trap doors, corrupt states …
- Invariant violations in subclass
- Syntactic problems as used in mutation testing
- Combinatorial testing

- Domain-specific faults

Mutation Analysis: Assessing Tests

- Apply small syntactical changes to a program
  - Modified program is called a mutant
  - One change per program
- Run existing test suite and count number of detected failures
  - If a test fails on a mutant, the mutant is killed
- If a test suite does not kill all mutants that are representative of a specific class of faults, this may hint at deficiencies of the test suite
  - Add tests!
  - Mutation score is #mutants killed/#non-equivalent mutants

Method-Level Mutation Operators

Possibly plus off-by-constant, wrong variable, … [Ma et al. 02]

Inter-Class Level Mutation Operators

Operators	Description
AMC	Access modifier change
IHD	Hiding variable deletion
IHI	Hiding variable insertion
IOD	Overriding method deletion
IOP	Overridden method calling position change
IOR	Overridden method rename
ISK	*super* keyword deletion
IPC	Explicit call of a parent's constructor deletion
PNC	*new* method call with child class type
PMD	Instance variable declaration with parent class type
PPD	Parameter variable declaration with child class type
PRV	Reference assignment with other compatible type
OMR	Overloading method contents change
OMD	Overloading method deletion
OAO	Argument order change
OAN	Argument number change
JTD	*this* keyword deletion
JSC	*static* modifier change
JID	Member variable initialization deletion
JDC	Java-supported default constructor create
EOA	Reference assignment and content assignment replacement
EOC	Reference comparison and content comparison replacement
EAM	Accessor method change
EMM	Modifier method change

[Ma et al.'02]

Faults and Mutants

Faults	Class Mutation Operators
State visibility anomaly	IOP
State definition inconsistency (due to state variable hiding)	IHD, IHI
State definition anomaly (due to overriding)	IOD
Indirect inconsistent state definition	IOD
Anomalous construction behavior	IOR, IPC, PNC
Incomplete construction	JID, JDC
Inconsistent type use	PID, PNC, PPD, PRV
Overloading methods misuse	OMD, OAO, OAN
Access modifier misuse	AMC
static modifier misuse	JSC
Incorrect overloading methods implementation	OMR
super keyword misuse	ISK
this keyword misuse	JTD
Faults from common programming mistakes	EOA, EOC, EAM, EMM

[Ma et al. 02]

Assumptions

- Competent Programmer Hypothesis
  "Programmers' faults are of a syntactical nature only"
  - Requirements ambiguous/incorrect?
  - Requirements misunderstood?
  - Inadequate assumptions on environment?
- Coupling hypothesis
  - "Syntactic faults correlate with other kinds of faults"
  - "Test suites that detect simple faults also detect complex faults"
    - Simple: can be corrected by changing one statement; complex: not simple
  - Some empirical evidence available [Offutt'92, Andrews'05]. I don't buy it.

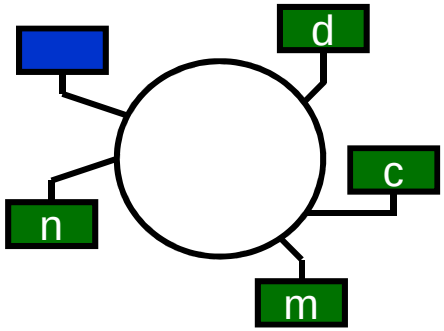
Fault models

- Limit testing
- Deadlocks, order violations, atomicity violations
- Incorrect transition, sneak paths, trap doors, corrupt states …
- Invariant violations in subclass
- Syntactic problems as used in mutation testing
- Combinatorial testing

- Domain-specific faults

Example: Automotive Infotainment Network

- DVD player: {d1,d2,d3,-}
- CD player: {c1,c2,c3,c4,-}
- Mobile phone: {m1,m2,m3,m4,m5,-}
- Navigation system: {n1,n2,-}



- Altogether, we have 4*5*6*3 possibilities
- Can we cut down this number?

Combinatorial Testing

- Assumption:
  Failures are a result of combinations of two (three, four, …) parameters, *not of all combinations*
- Let’s concentrate on these (comparatively few) cases

More abstractly

- With p parameters, each of which can take n values, we have n^p combinations
- Assuming that failures are not a result of the interaction of *all parameter values* but rather of an interaction of only *2* (3, 4, …) parameters, we can reduce the test set to pairwise (t-wise) interactions

- 3 parameters, all take values from {1,2,3}: 27 combinations
- Instead, let's test only 9 of them:
  (1,1,1),(1,2,2),(1,3,3),(2,1,3),(2,2,1),(2,3,2),(3,1,2),(3,2,3),(3,3,1)

- **Make sure this fault model is applicable when using it!**

Fault models [Morell 1991, Pretschner et al. 2013]

- Faults are delta with correct programs
- Fault models are descriptions of mappings from correct to incorrect programs and/or characterizations of hypothesized failure domains
  - Combinatorial testing special case
  - Limit testing easier to grasp by failure domain
- „Effective“ fault models simple to define

Remember: Fault models

- Limit testing
- Deadlocks, order violations, atomicity violations
- Incorrect transition, sneak paths, trap doors, corrupt states …
- Invariant violations in subclass
- Syntactic problems as used in mutation testing
- Combinatorial testing

- Domain-specific faults

Example I: Legacy Business IT

- Recurring faults

Project P1

RPG:

- System state management
  - Variables not re-initialized between workflows
  - State kept in temp DB tables
- Hard-coded values
- Incorrect data types
- Too loose or too restrictive checks
- Arithmetic bugs
- …

Project P2

Cobol:

- System state management
  - Global variable reuse
- Hard-coded values
- Arithmetic bugs
- Too loose or too restrictive checks
- Incorrect data types

PowerBuilder:

- Variables not re-initialized between workflows

PL/SQL:

- Too loose or too restrictive checks

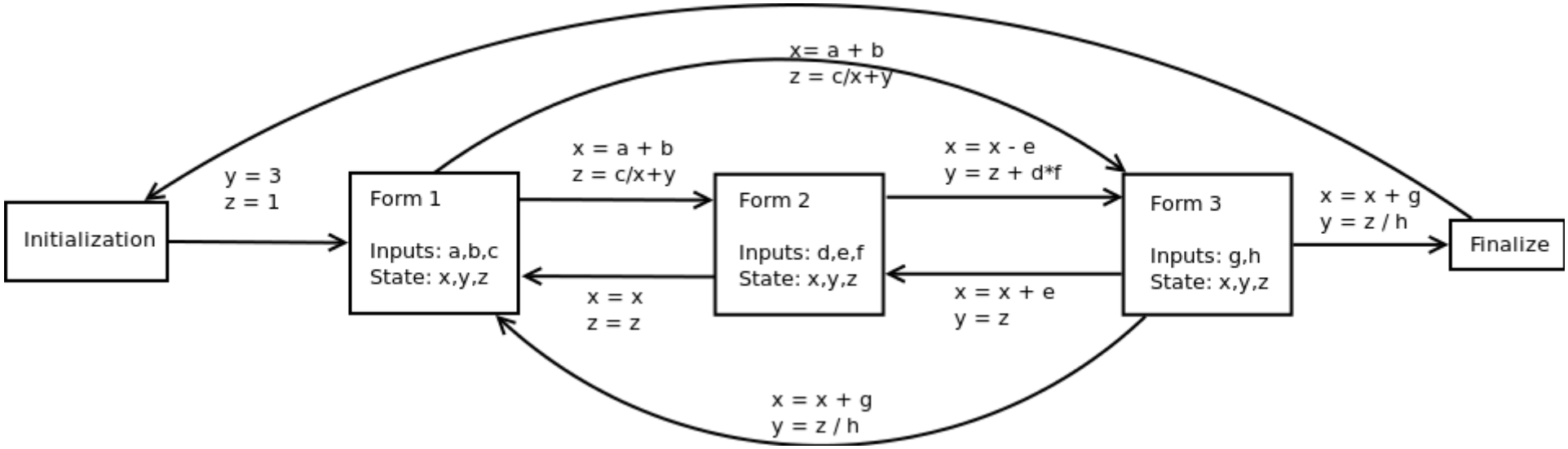
Aggregated View: Examples

Fault
Too loose or too restrictive checks / conditions
System state management (has sub categories)
Variables not re-initialized between workflows
Global variable reuse
State kept in temporary DB table
Hard-coded values
Incorrect data types
Arithmetic bugs

- And so on …

Example: Unintended Workflows

- **Problem**: navigating between forms in different ways leads to different results (failures)
- **Idea**:
  - Compare operations performed between forms (states) in different workflows
  - Use only “Next” button in GUI to determine intended or correct workflow
  - Test un-intended workflows dynamically to find high severity failures

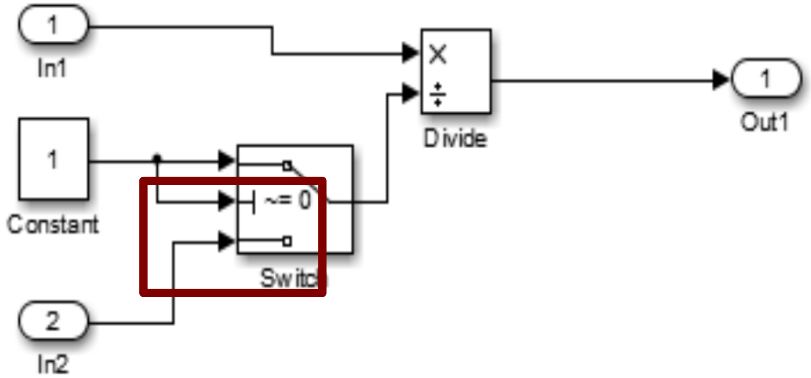


Example II: Continuous Systems

- Implementation of controllers in Matlab/Simulink
- Example 1
  over/underflows; division by zero (or close-to-zero)
  … using smells
- Example 2
  problems if intended value smaller than current value –
  usually, tests only for larger values

Fault Models for Matlab/Simulink Models: Faults

- Typical faults among others: over/underflows, scope violations, division by 0/almost 0
- Currently late detection at the end of the testing process with static analysis
- Cost!
- Optimization: Find this kind of faults early and at the unit level



Overflowing Abs – A Typical Fault Model


```
if x < 0:
        return -x
if x > 0:
        return x
```

Example: 8-bit signed integer

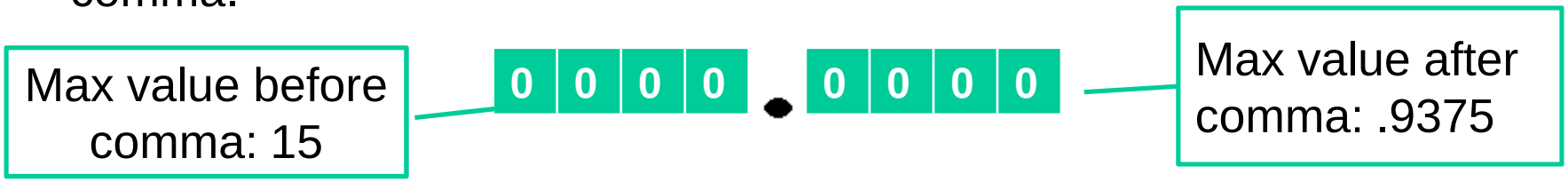
-128 = 10000000
(two complement)

Multiply with (-1) :
- Invert all bits -> 01111111
- Add 00000001

Result:
10000000 = -128
Once again!

Division by Small Value

- 8-bit unsigned fixed point value with 4 bits before and 4 after the comma.



Total value range: $15\frac{15}{16}, ..., 15\frac{1}{16}, 15, ..., \frac{15}{16}, ..., \frac{1}{16}, 0$

- Example:

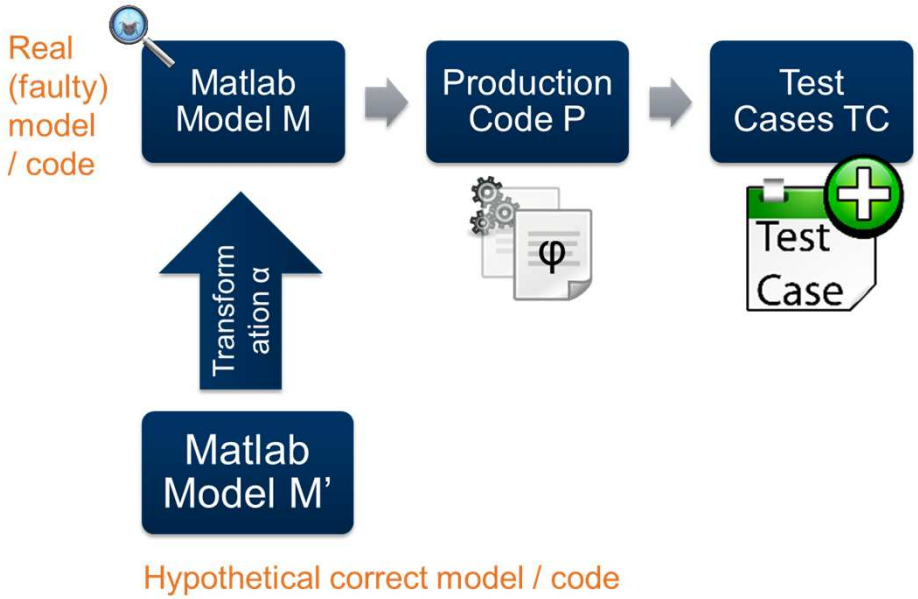
$$\frac{8}{\frac{1}{16}} = 8 \cdot 16 = 128$$

128 is far greater than the highest number (15.9375) that we could store in a 4/4-bit fixed point value

8 (decimal) as fixed point binary: 1000.0000, 1/16 as fix.p.bin: 0000.0001

Approach

- Analyze models for potential faults (smells)
- Derive and execute test as evidence for actual fault:
  Use potential faults to provoke failures
- Dynamic addition of further fault models

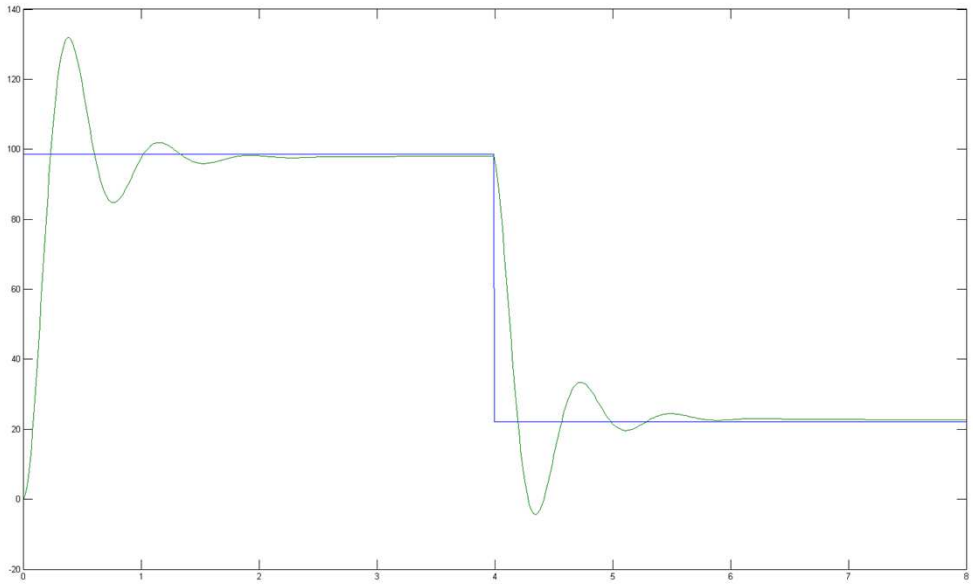


=> Early fault detection and direct localization in the model

Fault Model for Continuous Systems: Failures

- Complete test even more impossible than usual …
- Experts write representative tests
- Frequent assumption: controller is in initial state (that is, 0)
- Hence only „positive“ computations starting at 0

⇒ Sufficient to test requirements such as stability, responsiveness etc.?



Controller with strong undershooting

Approach

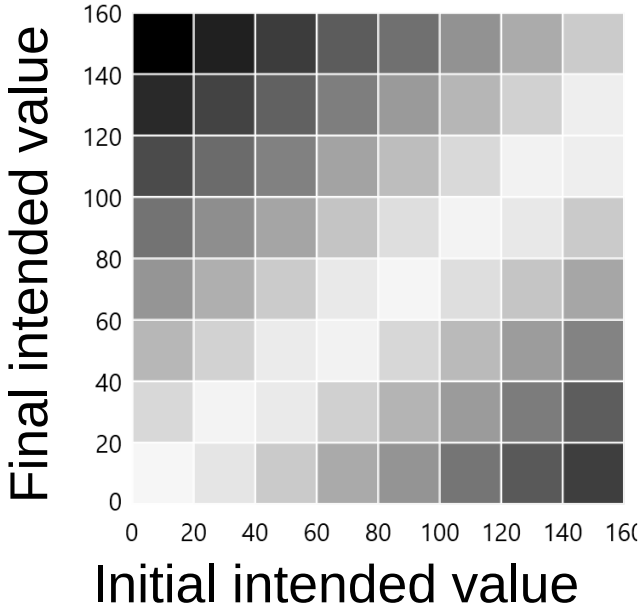
- Simulation with two intended values (fault model!)
  - First half: get system to initial intended value
  - Second half: get system to final intended value

Step 1 [Matinnejad et al. 2014]:

- Partition input space into blocks
- Randomly select N points per block
- Assess requirement satisfaction per point
- Create heatmap (brighter block = better satisfaction)

Step 2 [Matinnejad et al. 2014]:

- Use more fine-grained AI search methods for selected blocks
- Find global maximum of deviation for blocks

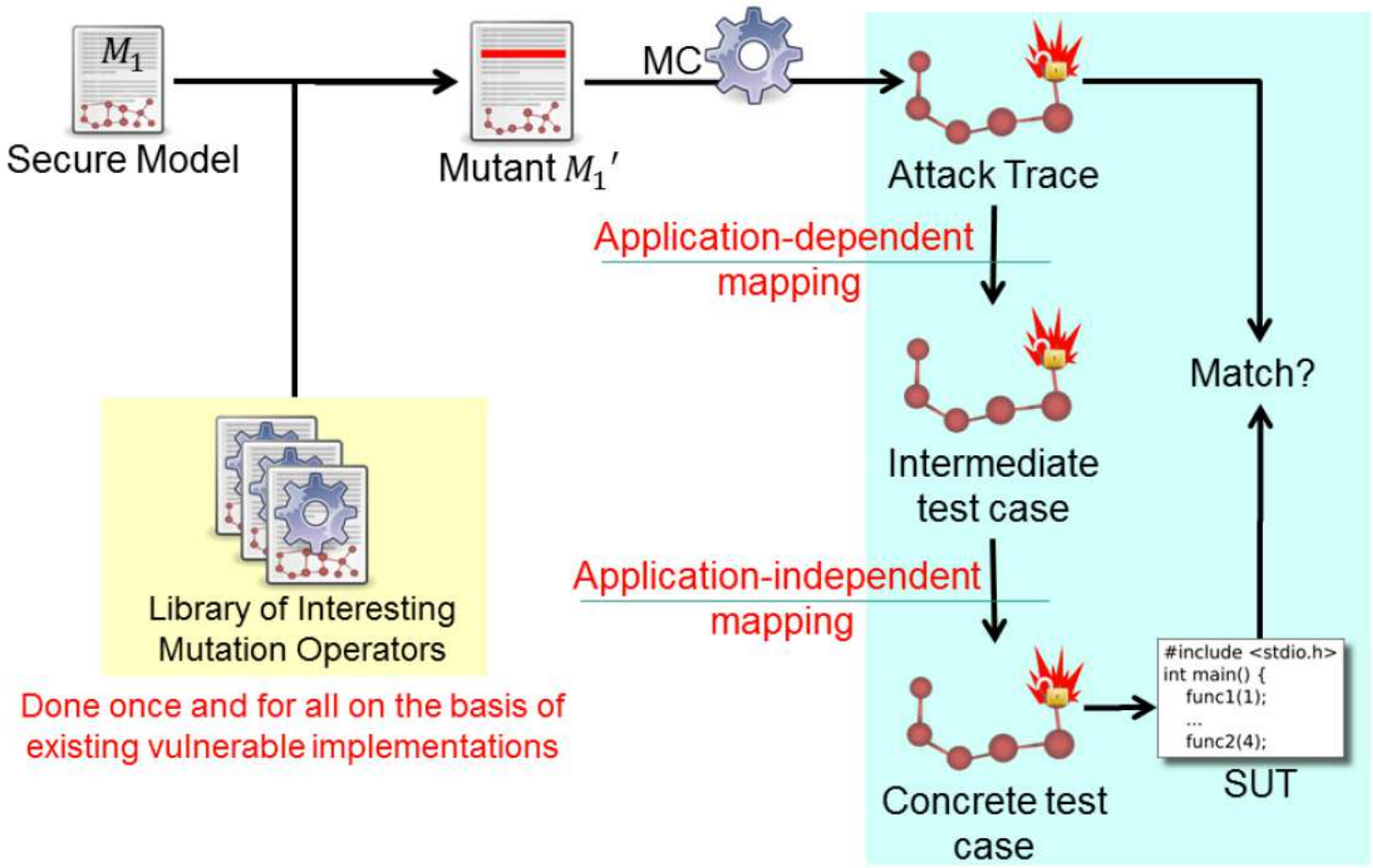


Stability heat map

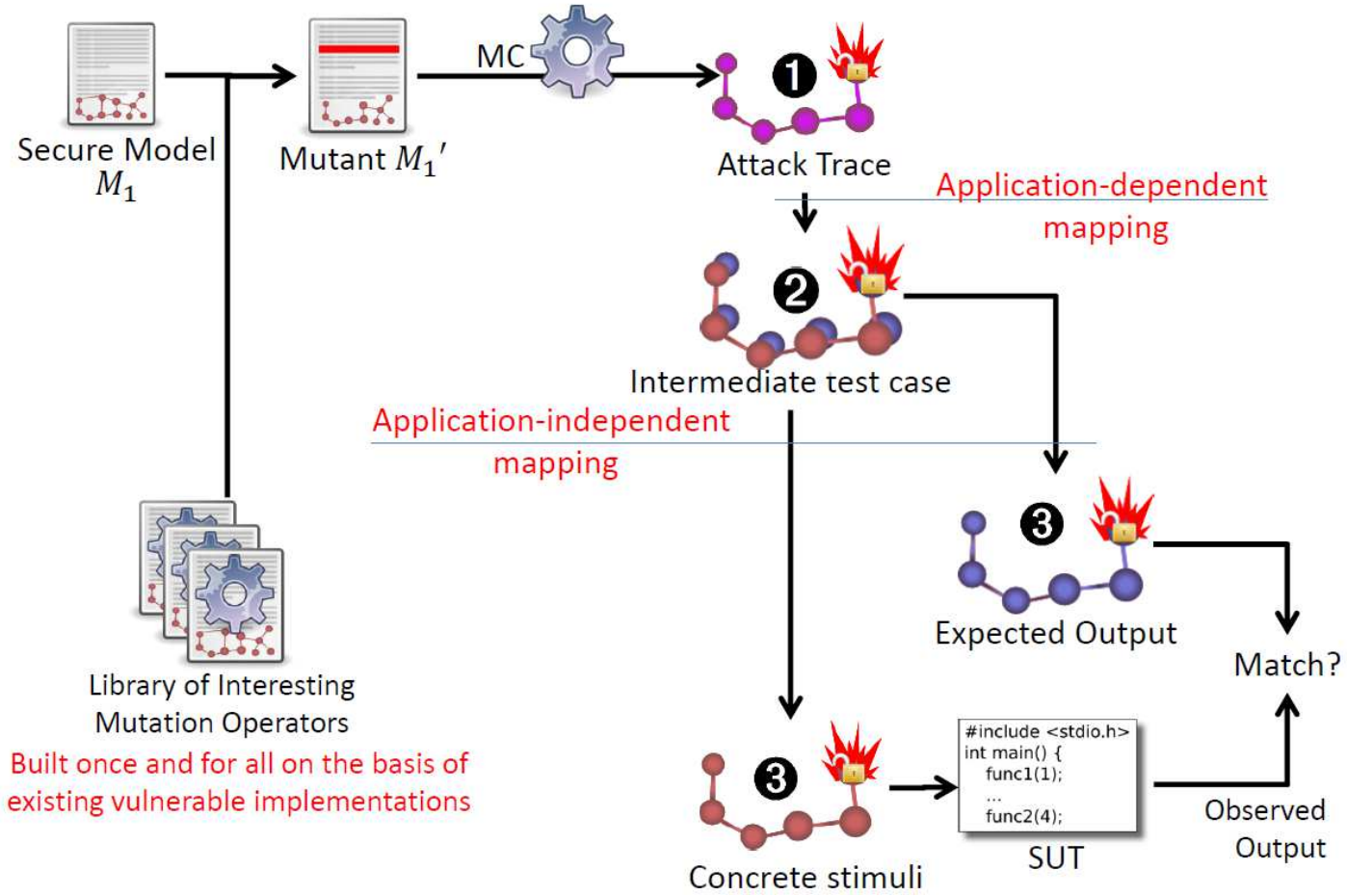
Further fault models, e.g. oscillation of plant after reaching intended value.
[Identifying these fault models is the crucial part!]

Example III: Security Tests for Web Apps

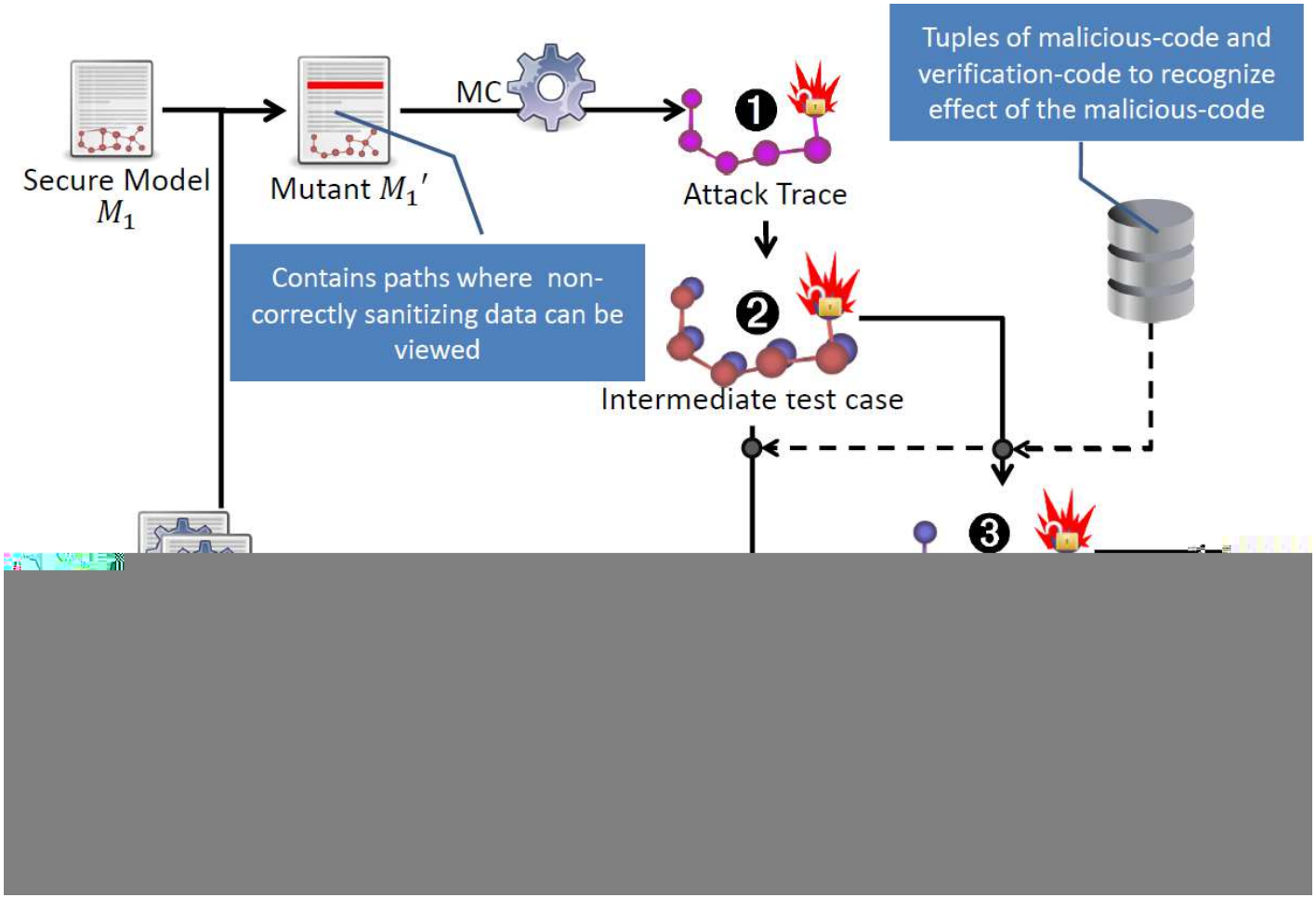
[Büchler et al. 2012]



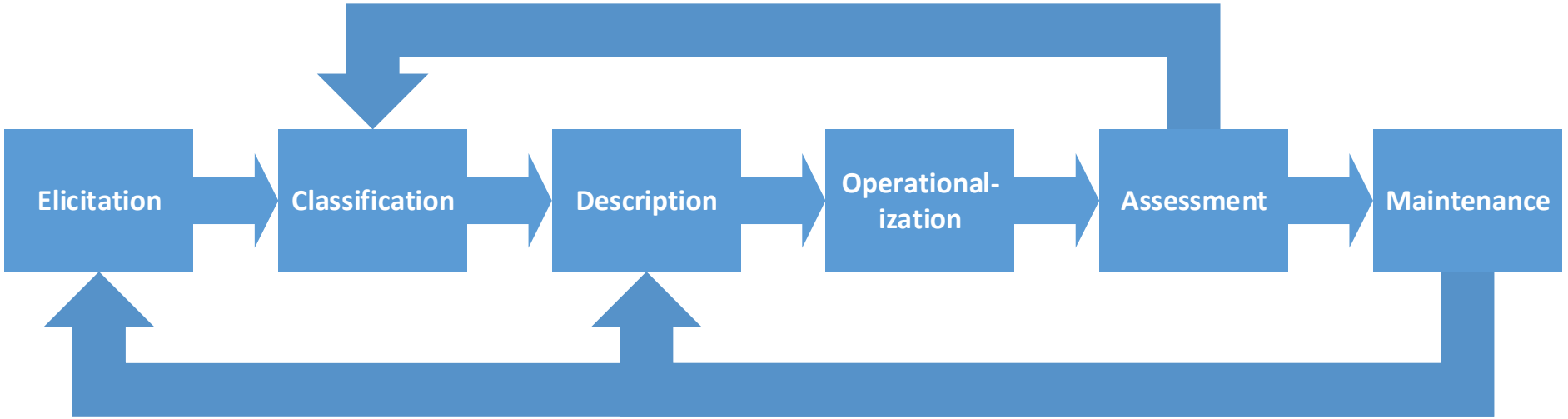
Example: XSS


Secure Model
M_1
Mutant M_1'
MC
1
Attack Trace
Application-dependent
mapping
2
Intermediate test case
Application-independent
mapping
3
Expected Output
Match?
Library of Interesting
Mutation Operators
Built once and for all on the basis of
existing vulnerable implementations
3
Concrete stimuli
#include <stdio.h>
int main() {
func1(1);
...
func2(4);
SUT
Observed
Output

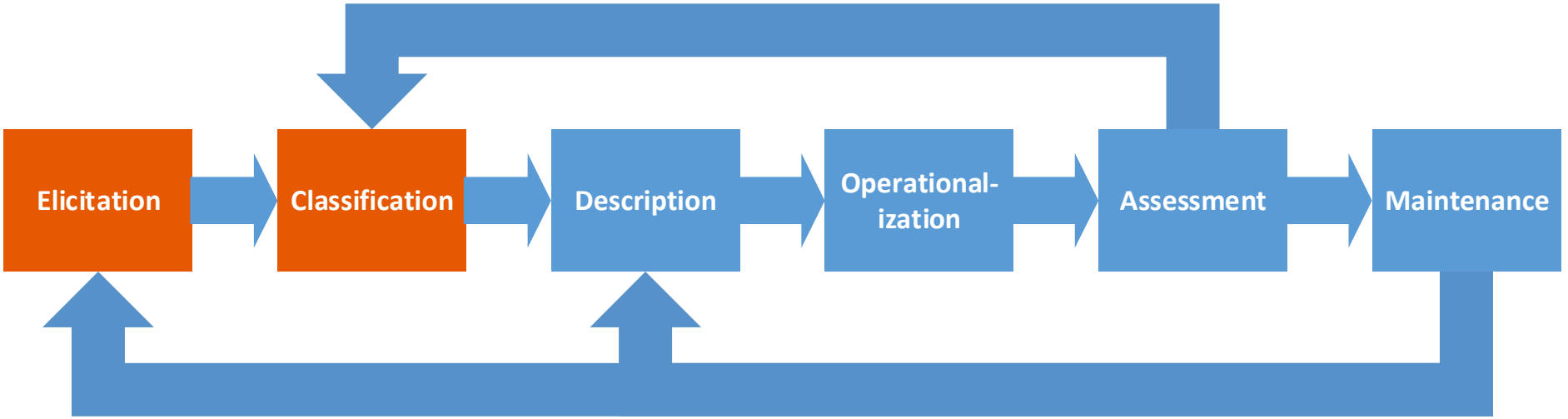
Example: XSS


Secure Model
M_1
Mutant M_1'
MC
Attack Trace
Tuples of malicious-code and verification-code to recognize effect of the malicious-code
Contains paths where non-correctly sanitizing data can be viewed
Intermediate test case
1
2
3

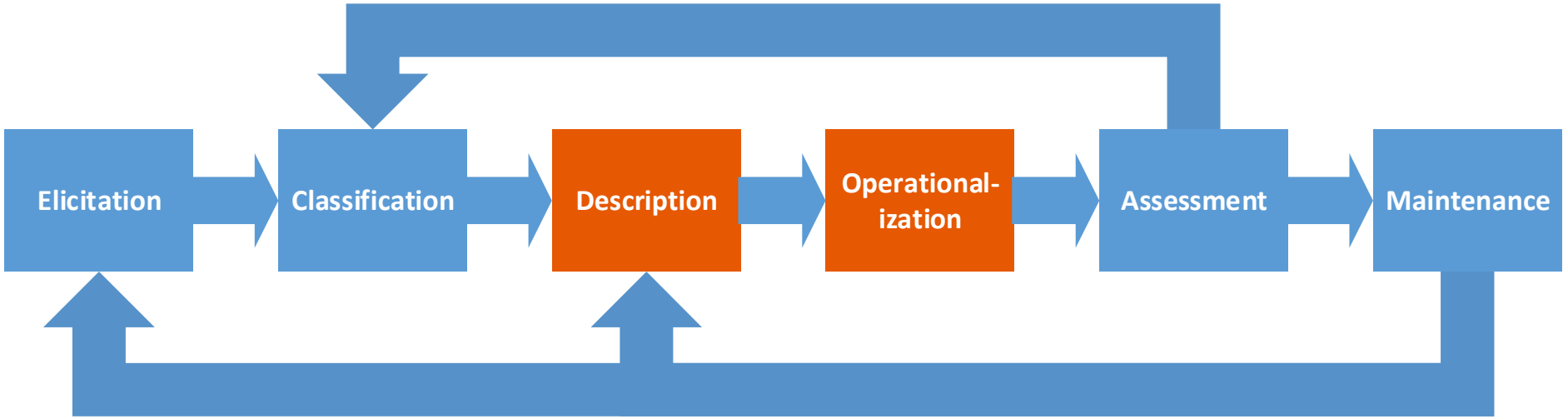
Process: Fault Model Lifecycle



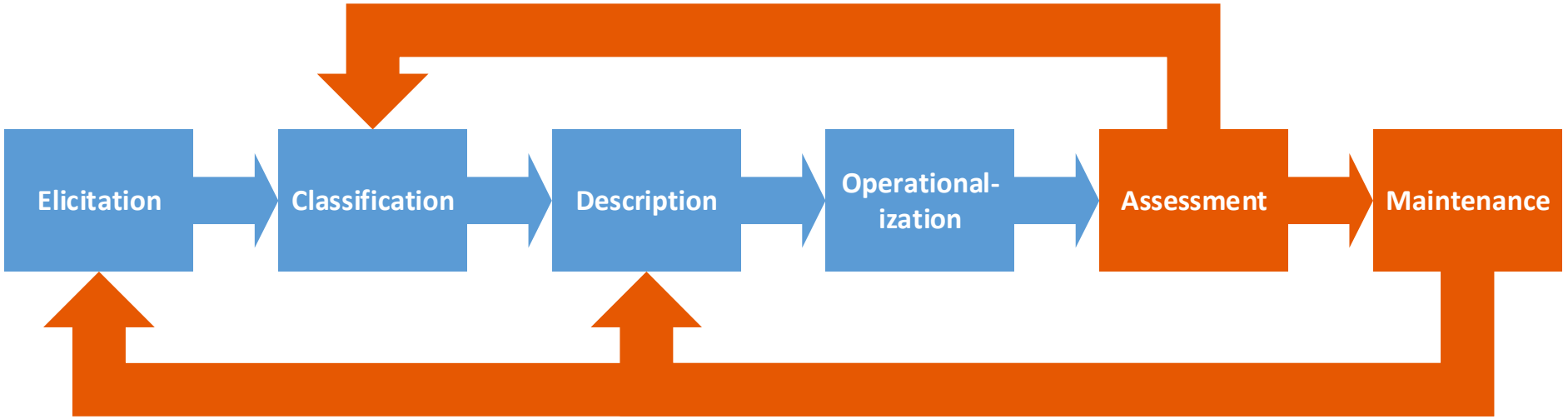
Process: Fault Model Lifecycle: Planning



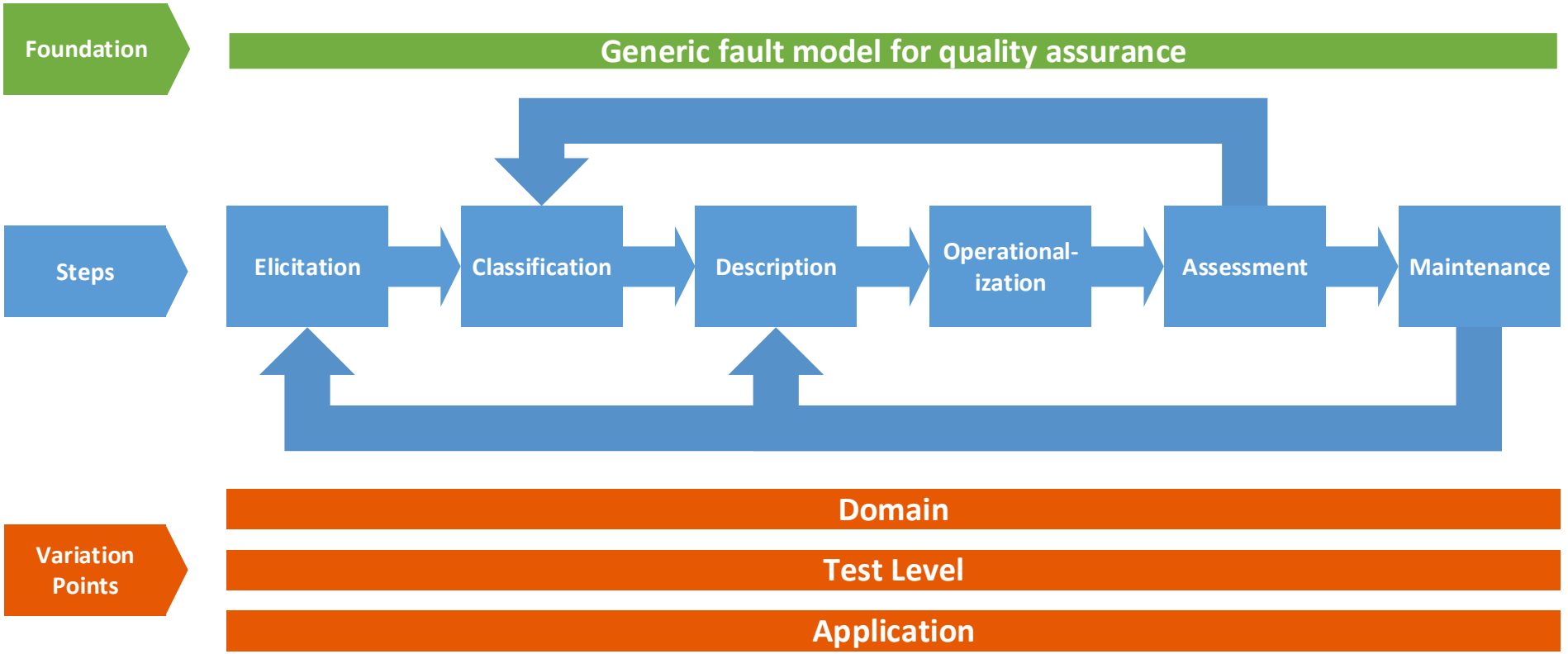
Process: Fault Model Lifecycle: Application



Process: Fault Model Lifecycle: Controlling



Process: Big Picture



Agenda

- Good tests?
- Partition-based testing
- Why coverage shouldn‘t be used a-priori
- Fault models
- Testing based on fault models
- Discussion

Test case derivation?

- Manually
  - You do that all the time!
- Automatically
  - Example security flaws
    - XSS, SQL injections, authentication flaws, …
    - Attacks based on fault models
  - Example Simulink models
  - Example UI testing
- Or checklists instead

Discussion

- Various tools do similar things – but for general faults
- Fault models available –
  code reading the more efficient approach?
- How much process, how much technology?
- How to build and maintain a good fault data base? Agility?
- Fault-based testing needs to be complemented!
- Fault injection not a new idea

(Deliberate) Limitations

```
. . .
hashOut.data = hashes + SSL_MD5_DIGEST_LEN;
hashOut.length = SSL_SHA1_DIGEST_LEN;
if ((err = SSLFreeBuffer(&hashCtx)) != 0)
    goto fail;
if ((err = ReadyHash(&SSLHashSHA1, &hashCtx)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &clientRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
    goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;

err = sslRawVerify(...);
. . .
```

Wrap-Up and Take-Home message

- „Good“ test cases require fault models
- Coverage not based on fault model
- „Fault models“ non-trivial
  - But everybody uses them all the time!
- Fault model needs to be applicable …
- … but not finding a problem doesn‘t make tests bad!!
- Operationalization: tests and check lists

- Continue to build a culture of faults!

1. OG DIREKT ÜBER DEM EMPFANG