On the Value of Fault Injection on the Modeling Level

Bernhard K. Aichernig

United Nations University
International Institute for Software Technology
Macao S.A.R. China

Overture Workshop 2005

Outline

Fault Injection

- Motivation: fault injection support in the Overture tools.
- Fault injection: insert faults and check the consequences
- Applications:
  - testing fault-tolerant systems
  - assessing test cases in programs
- Mutation testing:
  - inject faults into a program text (mutations)
  - assess existing test cases (mutants killed?)
  - design better test cases
  - introduced by Hamlet (1977) and DeMillo et al. (1978)
- Here: inject faults into a model (e.g. VDM++)

Fault Injection

- Motivation: fault injection support in the Overture tools.
- Fault injection: insert faults and check the consequences
- Applications:
  - testing fault-tolerant systems
  - assessing test cases in programs
- Mutation testing:
  - inject faults into a program text (mutations)
  - assess existing test cases (mutants killed?)
  - design better test cases
  - introduced by Hamlet (1977) and DeMillo et al. (1978)
- Here: inject faults into a model (e.g. VDM++)

Fault Injection

- Motivation: fault injection support in the Overture tools.
- Fault injection: insert faults and check the consequences
- Applications:
  - testing fault-tolerant systems
  - assessing test cases in programs
- Mutation testing:
  - inject faults into a program text (mutations)
  - assess existing test cases (mutants killed?)
  - design better test cases
  - introduced by Hamlet (1977) and DeMillo et al. (1978)
- Here: inject faults into a model (e.g. VDM++)

Fault Injection

- Motivation: fault injection support in the Overture tools.
- Fault injection: insert faults and check the consequences
- Applications:
    - testing fault-tolerant systems
    - assessing test cases in programs
- Mutation testing:
    - inject faults into a program text (mutations)
    - assess existing test cases (mutants killed?)
    - design better test cases
    - introduced by Hamlet (1977) and DeMillo et al. (1978)
- Here: inject faults into a model (e.g. VDM++)

From Errors via Faults to Failures

We follow the terminology of the IEEE Computer Society:

- An error is made by somebody. A good synonym is mistake. When people make mistakes during coding, we call these mistakes bugs.
- A fault is a representation of an error. As such it is the result of an error.
- A failure is a wrong behavior caused by a fault. A failure occurs when a fault executes.

From Errors via Faults to Failures

We follow the terminology of the IEEE Computer Society:

- An error is made by somebody. A good synonym is mistake. When people make mistakes during coding, we call these mistakes bugs.
- A fault is a representation of an error. As such it is the result of an error.
- A failure is a wrong behavior caused by a fault. A failure occurs when a fault executes.

From Errors via Faults to Failures

We follow the terminology of the IEEE Computer Society:

- An error is made by somebody. A good synonym is mistake. When people make mistakes during coding, we call these mistakes bugs.
- A fault is a representation of an error. As such it is the result of an error.
- A failure is a wrong behavior caused by a fault. A failure occurs when a fault executes.

Experience from Two Industrial Projects

- Domain: voice communication for air-traffic control
- Partners: FREQUENTIS and TU Graz.
- Project 1: VCS-3020S, a safety-critical radio and telephone switch (VDM++)
- Project 2: a network of VCS-3020Ss (VDM-SL)
- Aims:
  - Requirements specification and improvement.
  - Test-case evaluation through interpretation and coverage measures.

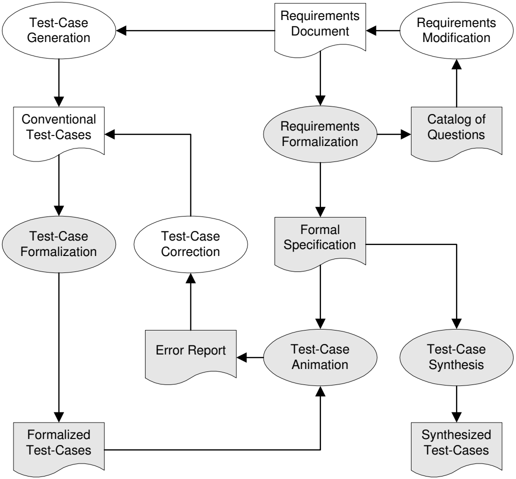
Experience from Two Industrial Projects

- Domain: voice communication for air-traffic control
- Partners: FREQUENTIS and TU Graz.
- Project 1: VCS-3020S, a safety-critical radio and telephone switch (VDM++)
- Project 2: a network of VCS-3020Ss (VDM-SL)
- Aims:
  - Requirements specification and improvement.
  - Test-case evaluation through interpretation and coverage measures.

Experience from Two Industrial Projects

- Domain: voice communication for air-traffic control
- Partners: FREQUENTIS and TU Graz.
- Project 1: VCS-3020S, a safety-critical radio and telephone switch (VDM++)
- Project 2: a network of VCS-3020Ss (VDM-SL)
- Aims:
  - Requirements specification and improvement.
  - Test-case evaluation through interpretation and coverage measures.

Method Overview



Requirements Validation

Requirement Issues

- missing requirements,
- ambiguous requirements, or even
- conflicting requirement descriptions.

Project 1

64 issues found during specification (2400 lines VDM++)

Project 2

108 issues found during specification, in 140 requirements, leading to 33 changes

Requirements Validation

Requirement Issues

- missing requirements,
- ambiguous requirements, or even
- conflicting requirement descriptions.

Project 1

64 issues found during specification (2400 lines VDM++)

Project 2

108 issues found during specification, in 140 requirements, leading to 33 changes

Requirements Validation

Requirement Issues

- missing requirements,
- ambiguous requirements, or even
- conflicting requirement descriptions.

Project 1

64 issues found during specification (2400 lines VDM++)

Project 2

108 issues found during specification, in 140 requirements, leading to 33 changes

Test Case Validation

Test Case Issues

- missing test cases,
- missing interactions (test steps) in a test case,
- wrong interactions (test steps),
- wrong input-data or predicted test results

Project 1

Internal system test cases covered only appr. 80% of the spec.

Project 2

- new acceptance test cases covered 100% of spec.
- 25% of test cases faulty, 16 faults in 65 cases (200 steps)

Test Case Validation

Test Case Issues

- missing test cases,
- missing interactions (test steps) in a test case,
- wrong interactions (test steps),
- wrong input-data or predicted test results

Project 1

Internal system test cases covered only appr. 80% of the spec.

Project 2

- new acceptance test cases covered 100% of spec.
- 25% of test cases faulty, 16 faults in 65 cases (200 steps)

Test Case Validation

Test Case Issues

- missing test cases,
- missing interactions (test steps) in a test case,
- wrong interactions (test steps),
- wrong input-data or predicted test results

Project 1

Internal system test cases covered only appr. 80% of the spec.

Project 2

- new acceptance test cases covered 100% of spec.
- 25% of test cases faulty, 16 faults in 65 cases (200 steps)

Lessons Learned

- Test cases enhance communication
  - engineers did not like to read VDM models but test cases
- Test cases need to be validated as well
  - high number of faults in test cases decreases trust in test documentation
  - validation via test execution or test generation
- Expression coverage not appropriate for assessing or generating test cases
  - size of model depends on skills of modeller
- Control-flow based coverage criteria do not help
  - control-flow could be different from implementation
  - level of abstraction?

Lessons Learned

- Test cases enhance communication
  - engineers did not like to read VDM models but test cases
- Test cases need to be validated as well
  - high number of faults in test cases decreases trust in test documentation
  - validation via test execution or test generation
- Expression coverage not appropriate for assessing or generating test cases
  - size of model depends on skills of modeller
- Control-flow based coverage criteria do not help
  - control-flow could be different from implementation
  - level of abstraction?

Lessons Learned

- Test cases enhance communication
  - engineers did not like to read VDM models but test cases
- Test cases need to be validated as well
  - high number of faults in test cases decreases trust in test documentation
  - validation via test execution or test generation
- Expression coverage not appropriate for assessing or generating test cases
  - size of model depends on skills of modeller
- Control-flow based coverage criteria do not help
  - control-flow could be different from implementation
  - level of abstraction?

Lessons Learned

- Test cases enhance communication
  - engineers did not like to read VDM models but test cases
- Test cases need to be validated as well
  - high number of faults in test cases decreases trust in test documentation
  - validation via test execution or test generation
- Expression coverage not appropriate for assessing or generating test cases
  - size of model depends on skills of modeller
- Control-flow based coverage criteria do not help
  - control-flow could be different from implementation
  - level of abstraction?

Test Case Generation by Fault Injection

- Fault-based Testing focuses on faults
- not on structural coverage (e.g. cover all statements)
- Basic method:
  1. anticipate faults
  2. design test cases that would uncover such faults
  3. run these tests to detect such faults
- we model faults on the specification level
- by mutating the specification text

Questions

Interesting questions when focusing on faults:

- Does an error made by a designer or programmer lead to an observable fault?
- Do my test cases detect such faults?
- How do I find a test case that uncovers a certain fault?
- What are the equivalent test cases that would uncover such a fault?
- How to automatically generate test cases that will reveal certain faults?

Test Hypothesis

Assumption

- We can anticipate the errors possibly made during implementation and
- are able to represent the faults in a given model

Dijkstra

Testing can never show the absence of faults but only their presence.

Our Reply

Testing can show the absence of faults, if we have a knowledge of what can go wrong.

Test Hypothesis

Assumption

- We can anticipate the errors possibly made during implementation and
- are able to represent the faults in a given model

Dijkstra

Testing can never show the absence of faults but only their presence.

Our Reply

Testing can show the absence of faults, if we have a knowledge of what can go wrong.

Test Hypothesis

Assumption

- We can anticipate the errors possibly made during implementation and
- are able to represent the faults in a given model

Dijkstra

Testing can never show the absence of faults but only their presence.

Our Reply

Testing can show the absence of faults, if we have a knowledge of what can go wrong.

Fault Injection

- Error guessing is a common strategy of testers who are domain experts.
- Claim: with a model more systematic way of error guessing
- Kinds of fault injection:
  - automatically by a set of mutation operators,
  - manually by interactively altering the specification.

Fault Injection

- Error guessing is a common strategy of testers who are domain experts.
- Claim: with a model more systematic way of error guessing
- Kinds of fault injection:
  - automatically by a set of mutation operators,
  - manually by interactively altering the specification.

Triangle Example: Original

```
context Ttype(a: int, b: int, c: int): String
pre:  a >= 1 and b >= 1 and c >= 1 and
      a < (b+c) and b < (a+c) and c < (a+b)
post: if((a = b) and (b = c))
      then result = "equilateral"
      else
        if ((a = b) or (a = c) or (b = c))
        then result = "isosceles"
        else result = "scalene"
        endif
      endif
```

Test case

$$a = 2,\ b = 2,\ c = 1,\ result = \text{“}isosceles\text{”}$$

Triangle Example: Mutant

```
context Ttype(a: int, b: int, c: int): String
pre:  a >= 1 and b >= 1 and c >= 1 and
      a < (b+c) and b < (a+c) and c < (a+b)
post: if((a = a) and (b = c))
      then result = "equilateral"
      else
        if ((a = b) or (a = c) or (b = c))
        then result = "isosceles"
        else result = "scalene"
        endif
      endif
```

Fault-detecting test case

$$a = 1,\ b = 2,\ c = 2,\ result = \text{“}isosceles\text{”}$$

Test Case Generation Algorithm

Given $D(Pre \vdash Post)$ and $D^m(Pre^m \vdash Post^m)$.

1. A test case T is searched by
   1. looking for a pair (i_c, O_c) being a solution of
   $$Pre \wedge Post^m \wedge \neg Post$$
   2. If it exists, then the test case $T = t(i, O)$ is generated by finding a maximal solution (i, O) of
   $$Pre \wedge Post \wedge (v = i_c)$$

Test Case Generation Algorithm

Given $D(Pre \vdash Post)$ and $D^m(Pre^m \vdash Post^m)$.

1. A test case T is searched by
    1. looking for a pair (i_c, O_c) being a solution of

    $$Pre \wedge Post^m \wedge \neg Post$$

    2. If it exists, then the test case $T = t(i, O)$ is generated by finding a maximal solution (i, O) of

    $$Pre \wedge Post \wedge (v = i_c)$$

Test Case Generation Algorithm(cont.)

2. If the former does not succeed, then we look for a test case $T = t(i, O)$ with (i, O) being a maximal solution of

$$\neg Pre^m \wedge Pre \wedge Post$$

If no test case has been found at this point, D and D^m are equivalent in the finite domain of the CSP.

Test Case Generation Algorithm(cont.)

2. If the former does not succeed, then we look for a test case $T = t(i, O)$ with (i, O) being a maximal solution of

$$\neg Pre^m \wedge Pre \wedge Post$$

If no test case has been found at this point, D and D^m are equivalent in the finite domain of the CSP.

Tool

- Algorithm is proved to be correct (in our testing theory).
- Implemented as a constraint solver
- currently OCL as input language
- Frühwirt's Constrain Handling Rules (CHR) to implement solver (Java)
- plus simplifications by generating the disjunctive normal form (DNF)
- interactive or automatic mutant generation
- alternatively, test case generation by DNF partitioning (Dick & Faivre 91)

Tool

- Algorithm is proved to be correct (in our testing theory).
- Implemented as a constraint solver
- currently OCL as input language
- Frühwirt's Constrain Handling Rules (CHR) to implement solver (Java)
- plus simplifications by generating the disjunctive normal form (DNF)
- interactive or automatic mutant generation
- alternatively, test case generation by DNF partitioning (Dick & Faivre 91)

Tool

- Algorithm is proved to be correct (in our testing theory).
- Implemented as a constraint solver
- currently OCL as input language
- Frühwirt's Constrain Handling Rules (CHR) to implement solver (Java)
- plus simplifications by generating the disjunctive normal form (DNF)
- interactive or automatic mutant generation
- alternatively, test case generation by DNF partitioning (Dick & Faivre 91)

Fault-based vs. DNF

- Generating the DNF partitions and picking one test case out of each partition:

DNF-based test cases

$a = 2, b = 2, c = 1, result = isosceles$
$a = 2, b = 3, c = 4, result = scalene$
$a = 1, b = 1, c = 1, result = equilateral$
$a = 2, b = 1, c = 2, result = isosceles$
$a = 1, b = 2, c = 2, result = isosceles$

- The previous mutation would have been caught!

Fault-based vs. DNF

- Generating the DNF partitions and picking one test case out of each partition:

DNF-based test cases

$a = 2, b = 2, c = 1, result = isosceles$
$a = 2, b = 3, c = 4, result = scalene$
$a = 1, b = 1, c = 1, result = equilateral$
$a = 2, b = 1, c = 2, result = isosceles$
$a = 1, b = 2, c = 2, result = isosceles$

- The previous mutation would have been caught!

Fault-based vs. DNF

- Generating the DNF partitions and picking one test case out of each partition:

DNF-based test cases

$a = 2, b = 2, c = 1, result = isosceles$
$a = 2, b = 3, c = 4, result = scalene$
$a = 1, b = 1, c = 1, result = equilateral$
$a = 2, b = 1, c = 2, result = isosceles$
$a = 1, b = 2, c = 2, result = isosceles$

- The previous mutation would have been caught!

Fault-based vs. DNF (cont.)

```
context Ttype(a: int, b: int, c: int): String
pre:  a >= 1 and b >= 1 and c >= 1 and
      a < (b+c) and b < (a+c) and c < (a+b)
post: if((a = b) and (b = 1))
      then result = "equilateral"
      else
        if ((a = b) or (a = c) or (b = c))
        then result = "isosceles"
        else result = "scalene"
        endif
      endif
```

Fault-detecting Test case

$$a = 2,\ b = 2,\ c = 2,\ result = \text{“}equilateral\text{”}$$

Fault-based vs. DNF (cont.)

```
context Ttype(a: int, b: int, c: int): String
pre:  a >= 1 and b >= 1 and c >= 1 and
      a < (b+c) and b < (a+c) and c < (a+b)
post: if((a = b) and (b = c))
      then result = "equilateral"
      else
        if ((a = b) or (a = c) or (b = 2))
        then result = "isosceles"
        else result = "scalene"
        endif
      endif
```

Fault-detecting Test case

$$a = 1,\ b = 3,\ c = 3,\ \mathit{result} = \text{“}\mathit{isosceles}\text{”}$$

Testing Executable Models

- larger models are faulty: in a recent project, 319 test cases found 28 faults in an RSL specification
- checking the testing efforts via
- injecting faults into
  - function/method bodies
  - invariant, pre-postcondition contracts (e.g. weakening)
- similar to program mutation testing
- RAISE tool supports (emacs interface)
  - interactive fault injection
  - test execution
  - result comparison

Testing Executable Models

- larger models are faulty: in a recent project, 319 test cases found 28 faults in an RSL specification
- checking the testing efforts via
- injecting faults into
  - function/method bodies
  - invariant, pre-postcondition contracts (e.g. weakening)
- similar to program mutation testing
- RAISE tool supports (emacs interface)
  - interactive fault injection
  - test execution
  - result comparison

Testing Executable Models

- larger models are faulty: in a recent project, 319 test cases found 28 faults in an RSL specification
- checking the testing efforts via
- injecting faults into
  - function/method bodies
  - invariant, pre-postcondition contracts (e.g. weakening)
- similar to program mutation testing
- RAISE tool supports (emacs interface)
  - interactive fault injection
  - test execution
  - result comparison

Investigating Equivalent Mutants

- here one is interested in equivalent mutants
- not all injected faults lead to failures
- understanding the reasons increases understanding of the model
- equivalence or refinement checkers can be used
  - CADP Tool bisimulation checker (UNU-IIST)
  - FDR model checker for CSP (York)
- OCL test case generator: no test cases, if equivalent mutant

Investigating Equivalent Mutants

- here one is interested in equivalent mutants
- not all injected faults lead to failures
- understanding the reasons increases understanding of the model
- equivalence or refinement checkers can be used
  - CADP Tool bisimulation checker (UNU-IIST)
  - FDR model checker for CSP (York)
- OCL test case generator: no test cases, if equivalent mutant

Investigating Equivalent Mutants

- here one is interested in equivalent mutants
- not all injected faults lead to failures
- understanding the reasons increases understanding of the model
- equivalence or refinement checkers can be used
  - CADP Tool bisimulation checker (UNU-IIST)
  - FDR model checker for CSP (York)
- OCL test case generator: no test cases, if equivalent mutant

Conclusions

- Test cases are an essential tool in modeling
  - they are simple contracts, easily to communicate
  - testing the model and implementation
- Overture: put fault-based test case generation on the agenda
  - validating VDM++ models
  - supporting fault analysis, like in safety analysis
  - test cases to prevent faults in implementation
  - reducing bias in models via equivalent mutant analysis
- Future agenda:
  - Research: fault injection in computational and behavioural VDM++ specifications
  - Eclipse plug-ins: interactive mutator, constraint and/or SAT solvers

Conclusions

- Test cases are an essential tool in modeling
  - they are simple contracts, easily to communicate
  - testing the model and implementation
- Overture: put fault-based test case generation on the agenda
  - validating VDM++ models
  - supporting fault analysis, like in safety analysis
  - test cases to prevent faults in implementation
  - reducing bias in models via equivalent mutant analysis
- Future agenda:
  - Research: fault injection in computational and behavioural VDM++ specifications
  - Eclipse plug-ins: interactive mutator, constraint and/or SAT solvers

Conclusions

- Test cases are an essential tool in modeling
  - they are simple contracts, easily to communicate
  - testing the model and implementation
- Overture: put fault-based test case generation on the agenda
  - validating VDM++ models
  - supporting fault analysis, like in safety analysis
  - test cases to prevent faults in implementation
  - reducing bias in models via equivalent mutant analysis
- Future agenda:
  - Research: fault injection in computational and behavioural VDM++ specifications
  - Eclipse plug-ins: interactive mutator, constraint and/or SAT solvers

Conclusions

- Test cases are an essential tool in modeling
    - they are simple contracts, easily to communicate
    - testing the model and implementation
- Overture: put fault-based test case generation on the agenda
    - validating VDM++ models
    - supporting fault analysis, like in safety analysis
    - test cases to prevent faults in implementation
    - reducing bias in models via equivalent mutant analysis
- Future agenda:
    - Research: fault injection in computational and behavioural VDM++ specifications
    - Eclipse plug-ins: interactive mutator, constraint and/or SAT solvers