Model-based Mutation Testing

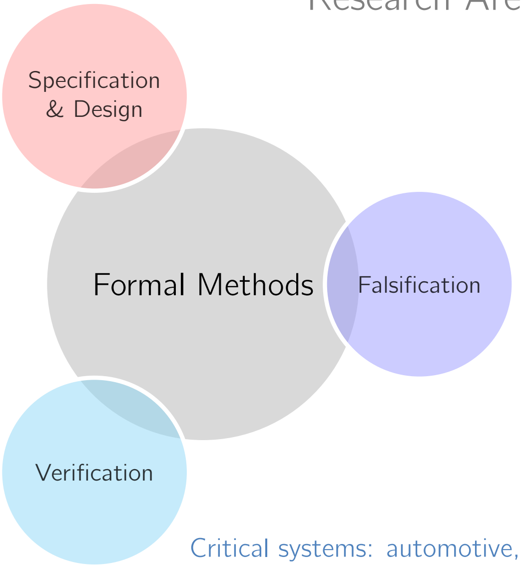
The Science of Killing Bugs in a Black Box

Bernhard K. Aichernig

Institute of Software Technology
Graz University of Technology
Austria

8th Halmstad Summer School on Testing, HSST 2018,
Halmstad University, 11 June 2018

Research Areas



- Precise abstraction
- Refinement
- Compositionality

- Model-based testing
- Model learning

- Proof-based development
- Statistical model checking

Critical systems: automotive, railways, IoT

FM Group Characteristics

- Size: key researcher + 3 research assistants (PhDs)
- EU projects: 4 in last 10 years
- LEAD project: Dependable Things
- Funding: EUR 192K per year (3 years avg.)
- Expertise: falsification + verification + languages
- Domains: automotive, railways, Internet of Things

DEPENDABLE INTERNET OF THINGS

TruConf

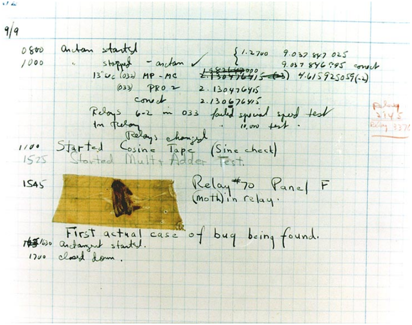
Agenda

- Mutation Testing
- Model-based Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
- Tool Demo and Examples

Bugs?

Part of engineering jargon for many decades:

- Moth trapped in relay of Mark II (Hopper 1946)
- Little faults and difficulties (Edison 1878):
- Software bugs



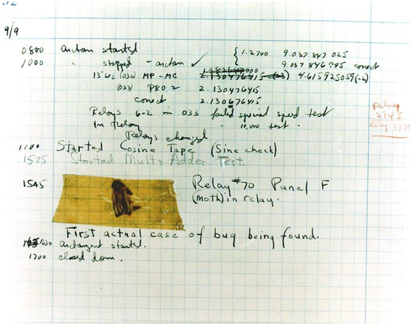
Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

Bugs?

Part of engineering jargon for many decades:

- Moth trapped in relay of Mark II (Hopper 1946)
- Little faults and difficulties (Edison 1878):
- Software bugs



Relay #70 Panel F
(moth) in relay.

First actual case of bug being found.

Bugs?

Part of engineering jargon for many decades:

- Moth trapped in relay of Mark II (Hopper 1946)
- Little faults and difficulties (Edison 1878):
- Software bugs

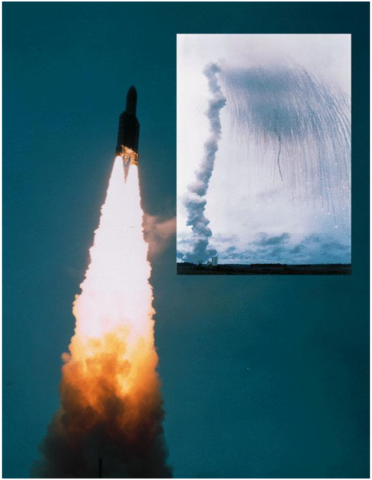
Definition

A software bug is the common term used to describe an

- error, flaw, mistake, failure, or fault in a computer program or system
- that produces an incorrect or unexpected result,
- or causes it to behave in unintended ways. (Wikipedia 2012)

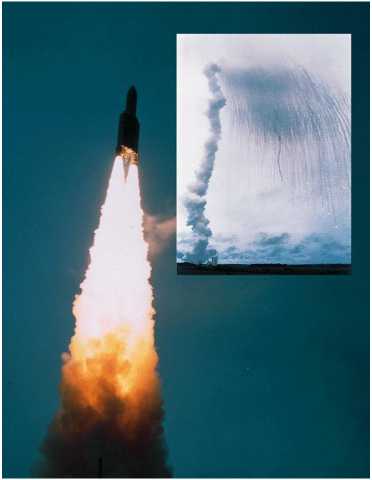
Some Bugs Become Famous!

- Ariane 5 test flight (1996)
  - out of control due to software failure
  - controlled destruction!
- Loss of
  - money and time
  - satellites
  - research (TU Graz)
- Dijkstra (EWD 1036):
  - call it error, not bug
  - a programmer created it



Some Bugs Become Famous!

- Ariane 5 test flight (1996)
  - out of control due to software failure
  - controlled destruction!
- Loss of
  - money and time
  - satellites
  - research (TU Graz)
- Dijkstra (EWD 1036):
  - call it error, not bug
  - a programmer created it



Some Bugs Hide for a Long Time!

Binary search bug in Java

- JDK 1.5 library (2006)
- out of boundary access of large arrays
- due to integer overflow
- 9 years undetected

```java
public static
int binarySearch(int[] a,int key)
{
  int low = 0;
  int high = a.length - 1;

  while (low <= high) {
    int mid = (low + high) / 2;
    int midVal = a[mid];

    if (midVal < key)
      low = mid + 1;
    else if (midVal > key)
      high = mid - 1;
    else
      return mid; // key found
  }
  return -(low + 1); // key not found
}
```

"Beware of bugs in the above code;
I have only proved it correct, not tried it."
[Knuth77]

Some Bugs Hide for a Long Time!

Binary search bug in Java

- JDK 1.5 library (2006)
- out of boundary access of large arrays
- due to integer overflow
- 9 years undetected

Algorithm was proven correct!

- Programming Pearls [Bentley86, Bentley00]
- assuming infinite integers :(

```
public static
int binarySearch(int[] a,int key)
{
  int low = 0;
  int high = a.length - 1;

  while (low <= high) {
    int mid = (low + high) / 2;
    int midVal = a[mid];

    if (midVal < key)
      low = mid + 1;
    else if (midVal > key)
      high = mid - 1;
    else
      return mid; // key found
  }
  return -(low + 1); // key not found
}
```

"Beware of bugs in the above code;
I have only proved it correct, not tried it."
[Knuth77]

Some Bugs Hide for a Long Time!

Binary search bug in Java

- JDK 1.5 library (2006)
- out of boundary access of large arrays
- due to integer overflow
- 9 years undetected

Algorithm was proven correct!

- Programming Pearls [Bentley86, Bentley00]
- assuming infinite integers :(

```
public static
int binarySearch(int[] a,int key)
{
  int low = 0;
  int high = a.length - 1;

  while (low <= high) {
    int mid = (low + high) / 2;
    int midVal = a[mid];

    if (midVal < key)
      low = mid + 1;
    else if (midVal > key)
      high = mid - 1;
    else
      return mid; // key found
  }
  return -(low + 1); // key not found
}
```

"Beware of bugs in the above code;
I have only proved it correct, not tried it."
[Knuth77]

Some Bugs Hide for a Long Time!

Binary search bug in Java

- JDK 1.5 library (2006)
- out of boundary access of large arrays
- due to integer overflow
- 9 years undetected

Algorithm was proven correct!

- Programming Pearls [Bentley86, Bentley00]
- assuming infinite integers :(

```java
public static
int binarySearch(int[] a,int key)
{
  int low = 0;
  int high = a.length - 1;

  while (low <= high) {
    int mid = (low + high) >>> 1;
    int midVal = a[mid];

    if (midVal < key)
      low = mid + 1;
    else if (midVal > key)
      high = mid - 1;
    else
      return mid; // key found
  }
  return -(low + 1); // key not found
}
```

"Beware of bugs in the above code;
I have only proved it correct, not tried it."
[Knuth77]

Observations

- Verification failed (wrong assumption)
- Established testing strategies failed:
  - statement coverage
  - branch coverage fails
  - multiple condition coverage
  - MC/DC: standard in avionics [DO-178B/ED109]
- Long array needed: `int[] a = new int[Integer.MAX_VALUE/2+2]`

Lesson

- Concentrate on possible faults, not on structure.
- Generate test cases covering these faults
- Mutation Testing [Lipton71, Hamlet77, DeMillo et al.78]

Observations

- Verification failed (wrong assumption)
- Established testing strategies failed:
  - statement coverage
  - branch coverage fails
  - multiple condition coverage
  - MC/DC: standard in avionics [DO-178B/ED109]
- Long array needed: `int[] a = new int[Integer.MAX_VALUE/2+2]`

Lesson

- Concentrate on possible faults, not on structure.
- Generate test cases covering these faults
- Mutation Testing [Lipton71, Hamlet77, DeMillo et al.78]

What Is Mutation Testing?

Originally: Technique to verify the quality of test cases

"There is a pressing need to address the, currently unresolved, problem of test case generation." [Jia&Harman11]

Y Jia, M Harman: *An analysis and survey of the development of mutation testing*. IEEE transactions on software engineering 37 (5), 2011.

What Is Mutation Testing?

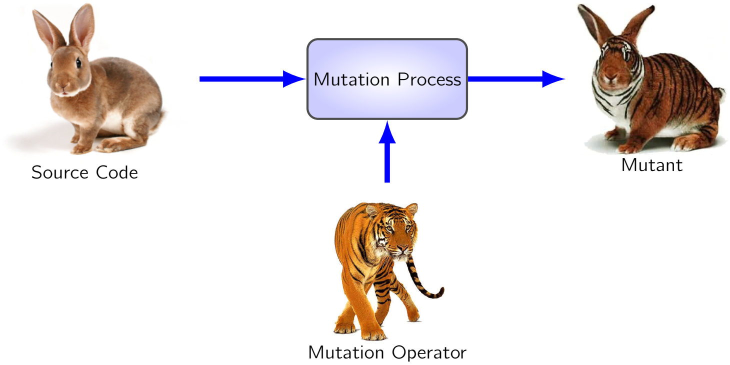
Originally: Technique to verify the quality of test cases

"There is a pressing need to address the, currently unresolved, problem of test case generation." [Jia&Harman11]

Y Jia, M Harman: *An analysis and survey of the development of mutation testing*. IEEE transactions on software engineering 37 (5), 2011.

How Does It Work?

Step 1: Create mutants



Example: Transformational System

- Kind of triangles:
  - equilateral
  - isosceles
  - scalene
- Create mutants
  - mutation operator
    == ⇒ >=
  - creates 5 mutants

```scala
object triangle {

 def tritype(a : Int, b : Int, c: Int) =
  (a,b,c) match {
  case _ if (a <= c-b) => "no triangle"
  case _ if (a <= b-c) => "no triangle"
  case _ if (b <= a-c) => "no triangle"
  case _ if (a == b && b == c) =>
                          "equilateral"
  case _ if (a == b) =>   "isosceles"
  case _ if (b == c) =>   "isosceles"
  case _ if (a == c) =>   "isosceles"
  case _  =>              "scalene"
  }
}
```

Source code in Scala

Example: Transformational System

- Kind of triangles:
  - equilateral
  - isosceles
  - scalene
- Create mutants
  - mutation operator
    == ⇒ >=
  - creates 5 mutants

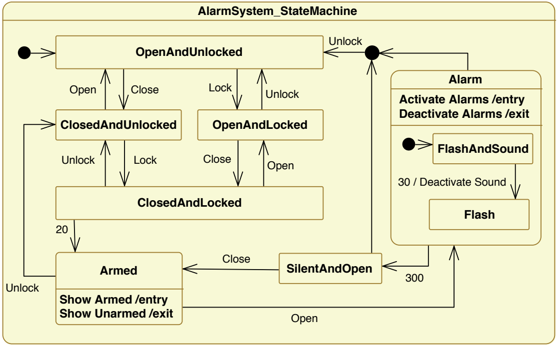
```
object triangle {

 def tritype(a : Int, b : Int, c: Int) =
  (a,b,c) match {
  case _ if (a <= c-b) => "no triangle"
  case _ if (a <= b-c) => "no triangle"
  case _ if (b <= a-c) => "no triangle"
  case _ if (a >= b && b == c) =>
                          "equilateral"
  case _ if (a == b) =>   "isosceles"
  case _ if (b == c) =>   "isosceles"
  case _ if (a == c) =>   "isosceles"
  case _  =>              "scalene"
  }
}
```

Mutant

Example: Reactive System

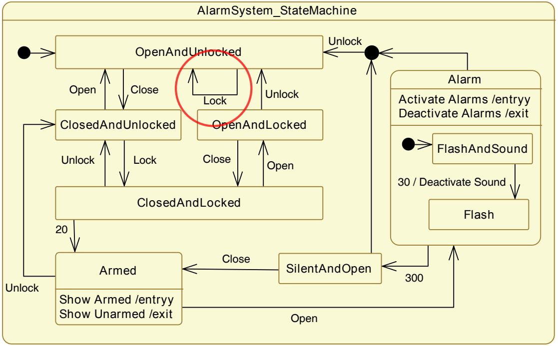
- Car Alarm System
  - event-based
  - controllable events
  - observable events
- Mutate the model
  - mutation operator
  - → ⇒ ↻
  - 17 mutants



State machine model in UML

Example: Reactive System

- Car Alarm System
  - event-based
  - controllable events
  - observable events
- Mutate the model
  - mutation operator
    $\longrightarrow \Rightarrow \circlearrowright$
  - 17 mutants



Mutated UML model

How Does It Work?

Step 2: Try to kill mutants

A test case kills a mutant if its run shows different behaviour.

Example: Transformational System

- Mutant survives path coverage (MC/DC):
  tritype(0,1,1)
  tritype(1,0,1)
  tritype(1,1,0)
  tritype(1,1,1)
  tritype(2,3,3)
  tritype(3,2,3)
  tritype(3,3,2)
  tritype(2,3,4)
- Mutant killed by
  tritype(3,2,2)

```
object triangle {

 def tritype(a : Int, b : Int, c: Int) =
  (a,b,c) match {
  case _ if (a <= c-b) => "no triangle"
  case _ if (a <= b-c) => "no triangle"
  case _ if (b <= a-c) => "no triangle"
  case _ if (a >= b && b == c) =>
                          "equilateral"
  case _ if (a == b) =>   "isosceles"
  case _ if (b == c) =>   "isosceles"
  case _ if (a == c) =>   "isosceles"
  case _  =>              "scalene"
  }
}
```

Mutant

Example: Transformational System

- Mutant survives path coverage (MC/DC):
  tritype(0,1,1)
  tritype(1,0,1)
  tritype(1,1,0)
  tritype(1,1,1)
  tritype(2,3,3)
  tritype(3,2,3)
  tritype(3,3,2)
  tritype(2,3,4)
- Mutant killed by
  tritype(3,2,2)

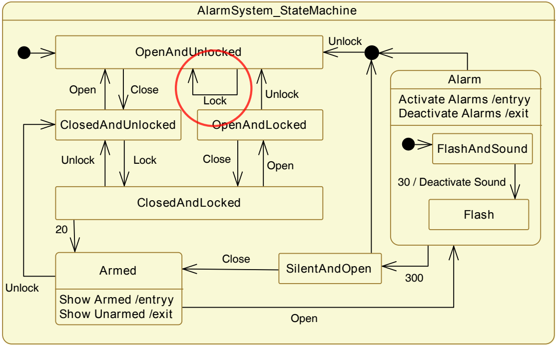
```
object triangle {

 def tritype(a : Int, b : Int, c: Int) =
  (a,b,c) match {
  case _ if (a <= c-b) => "no triangle"
  case _ if (a <= b-c) => "no triangle"
  case _ if (b <= a-c) => "no triangle"
  case _ if (a >= b && b == c) =>
                          "equilateral"
  case _ if (a == b) =>   "isosceles"
  case _ if (b == c) =>   "isosceles"
  case _ if (a == c) =>   "isosceles"
  case _  =>              "scalene"
  }
}
```

Mutant

Example: Reactive System

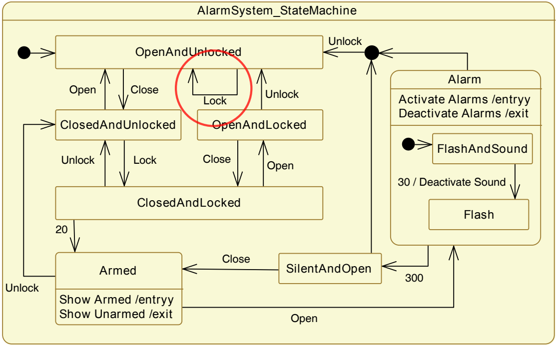
- Mutant survives
  - function coverage
  - state coverage
  - transition coverage
- Killed by
  Lock();
  Close();
  After(20);



Mutated UML model

Example: Reactive System

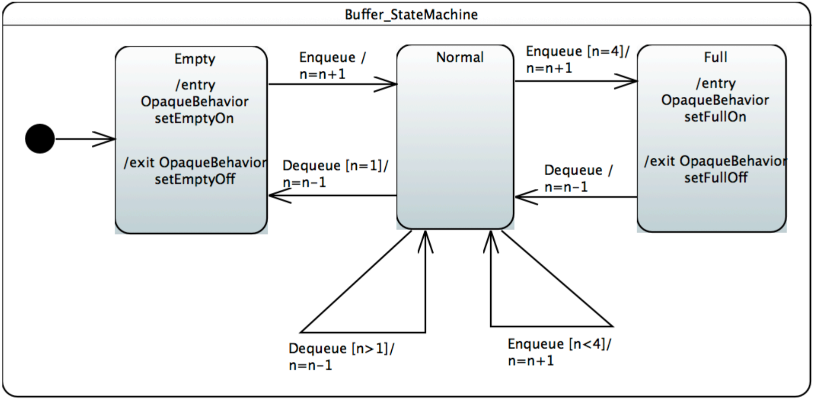
- Mutant survives
  - function coverage
  - state coverage
  - transition coverage
- Killed by
  Lock();
  Close();
  After(20);



Mutated UML model

Fault-Propagation in Models

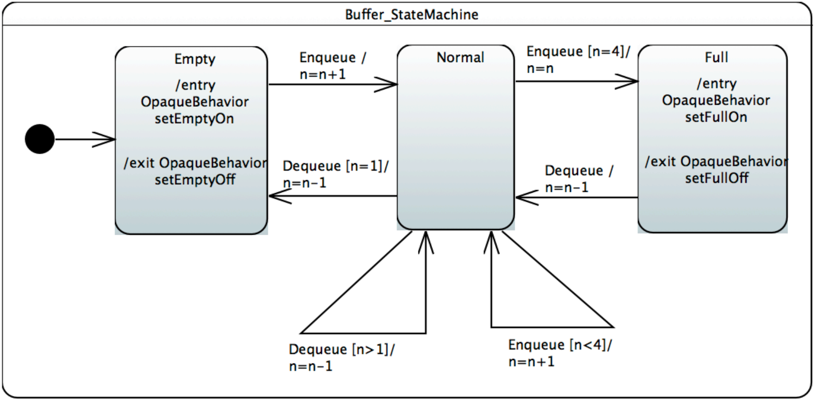
Abstract 5-place buffer model:



Counter variable n is internal!

Fault-Propagation in Models

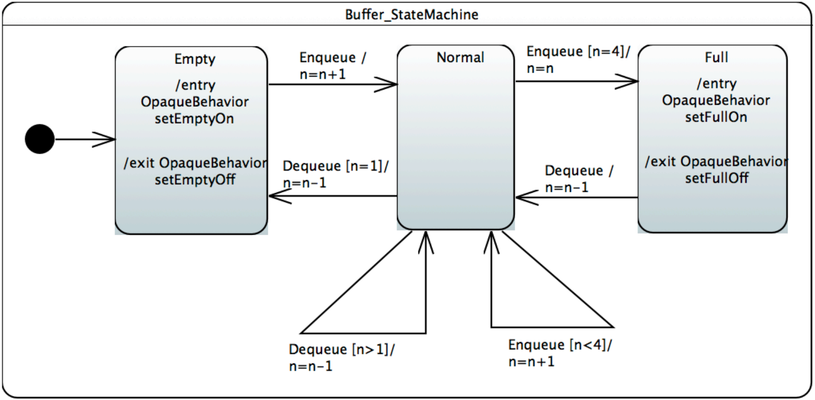
Let's inject a fault:



How does this fault propagate?

A Good Test Case

... triggers this fault and propagates it to a (visible) failure:



⟨!setEmptyOn, ?Enqueue, !setEmptyOff, ?Enqueue, ?Enqueue, ?Enqueue, ?Enqueue, !setFullOn, ?Dequeue, !setFullOff, ?Enqueue, !setFullOn⟩

From Analysis to Synthesis

State of art:

Analysis of test cases

How many mutants killed by test cases?

$$mutation\ score = \frac{\#killed\ mutants}{\#mutants}$$

Problem: equivalent mutants

Solution: review of surviving mutants

Research:

Synthesis of test cases

Find test cases that maximise mutation score.

Idea:

- Check equivalence between original and mutant
- Use counter-example as test case.

Problem: equivalence checking is hard (undecidable in general)

Solution: generate from models (abstraction)

→ model-based mutation testing

From Analysis to Synthesis

State of art:

Analysis of test cases

How many mutants killed by test cases?

$$mutation\ score = \frac{\#killed\ mutants}{\#mutants}$$

Research:

Synthesis of test cases

Find test cases that maximise mutation score.

Idea:

- Check equivalence between original and mutant
- Use counter-example as test case.

Problem: equivalent mutants

Solution: review of surviving mutants

Problem: equivalence checking is hard (undecidable in general)

Solution: generate from models (abstraction)

→ model-based mutation testing

From Analysis to Synthesis

State of art:

Analysis of test cases

How many mutants killed by test cases?

$$\textit{mutation score} = \frac{\#\textit{killed mutants}}{\#\textit{mutants}}$$

Problem: equivalent mutants

Solution: review of surviving mutants

Research:

Synthesis of test cases

Find test cases that maximise mutation score.

Idea:

- Check equivalence between original and mutant
- Use counter-example as test case.

Problem: equivalence checking is hard (undecidable in general)

Solution: generate from models (abstraction)

→ model-based mutation testing

From Analysis to Synthesis

State of art:

Analysis of test cases

How many mutants killed by test cases?

$$mutation\ score = \frac{\#killed\ mutants}{\#mutants}$$

Problem: equivalent mutants

Solution: review of surviving mutants

Research:

Synthesis of test cases

Find test cases that maximise mutation score.

Idea:

- Check equivalence between original and mutant
- Use counter-example as test case.

Problem: equivalence checking is hard (undecidable in general)

Solution: generate from models (abstraction)

→ model-based mutation testing

Agenda

- Mutation Testing
- Model-based Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
- Tool Demo and Examples

Model-based Testing

Model-based testing (MBT) is

- the automatic generation of software test procedures,
- using models of system requirements and behavior
- in combination with automated test execution.

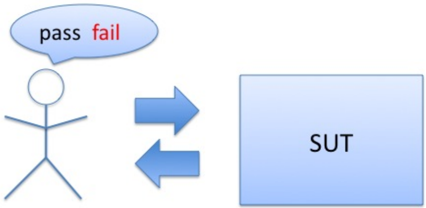
Objective

"Don't write test cases,

generate them!"

(John Hughes)

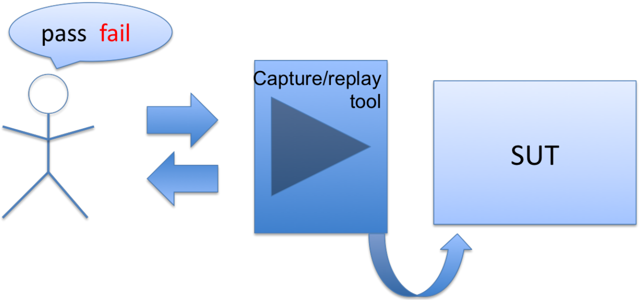
Levels of Testing: Manual



Levels of Testing: Manual

- \+ easy & cheap to start
- \+ flexible testing
- – expensive every execution
- – no auto regression testing
- – ad-hoc coverage
- – no coverage measurement

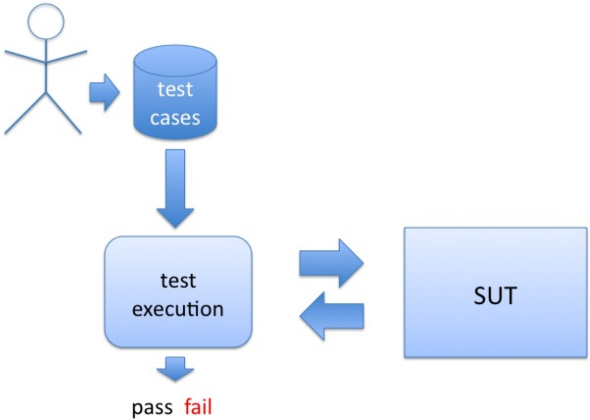
Levels of Testing: Capture & Replay



Levels of Testing: Capture & Replay

+ auto regression testing
+ flexible testing
– expensive first execution
– fragile tests break easily
– ad-hoc coverage
– no coverage measurement

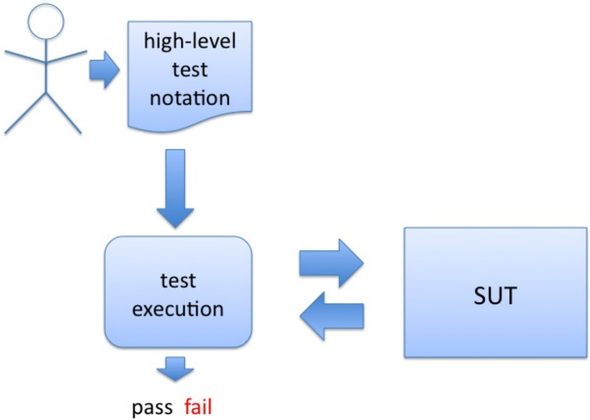
Levels of Testing: Scripts



Levels of Testing: Scripts

- + auto regression testing
- + automatic execution
- +/− test impl. = programming
- − fragile tests break easily? (depends on abstraction)
- − ad-hoc coverage
- − no coverage measurement

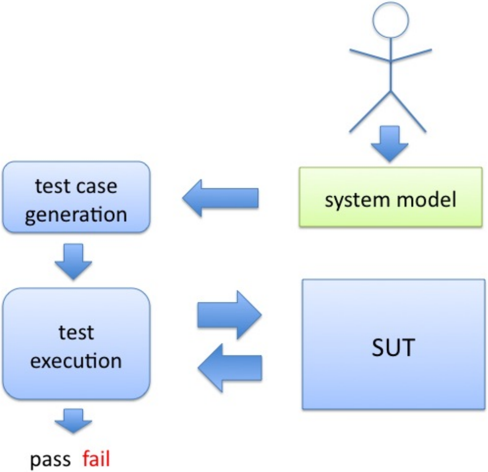
Levels of Testing: Test Scenarios



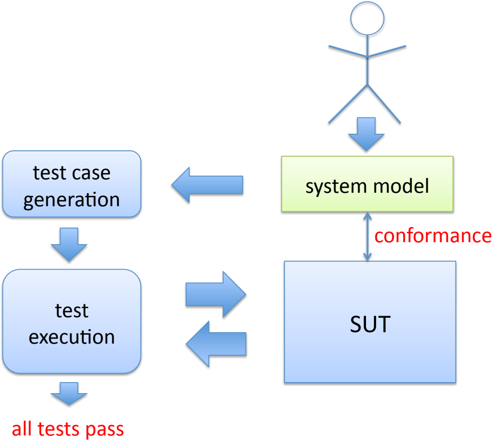
Levels of Testing: Test Scenarios

+ abstract tests
+ automatic execution
+ auto regression testing
+ robust tests
− ad-hoc coverage
− no coverage measurement

Levels of Testing: Model-Based Testing



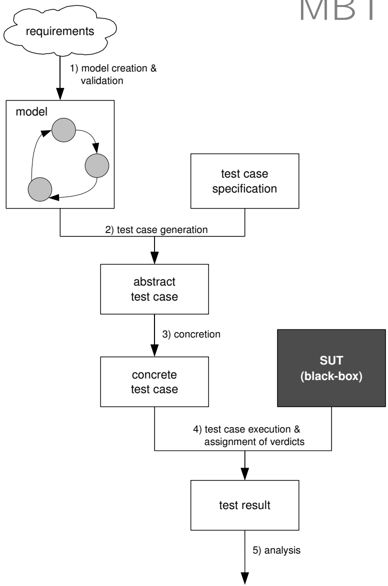
Levels of Testing: Model-Based Testing



Levels of Testing: Model-Based Testing

+ abstract tests
+ automatic execution
+ auto regression testing
+ auto design of tests
+ systematic coverage
+ measure coverage of model and requirements
– modelling efforts

MBT Workflow



Manual tasks:

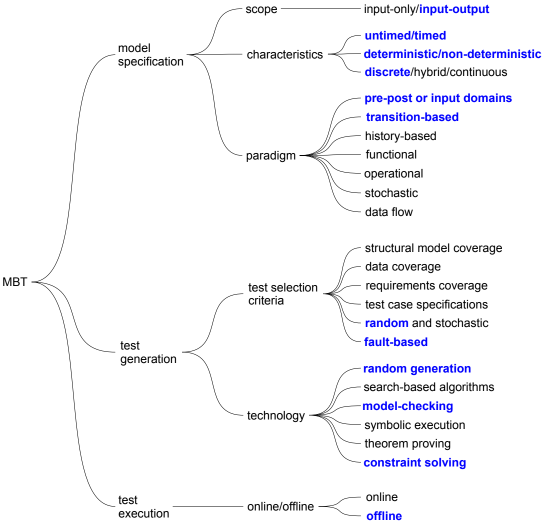
- (requirements analysis)
- model creation
- model validation
- concretion implementation

Automated tasks:

- model verification
- test-case generation
- test-case concretion
- test-case execution
- assignement of verdicts

Taxonomy

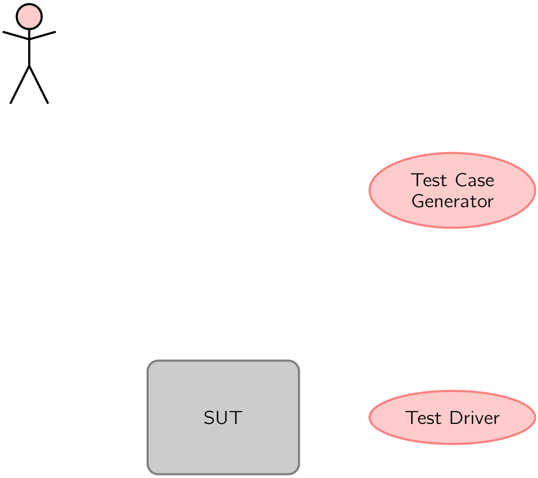
M. Utting, A. Pretschner, B. Legeard: *A taxonomy of model-based testing approaches.* Software Testing, Verification and Reliability, 22(5), 2012.



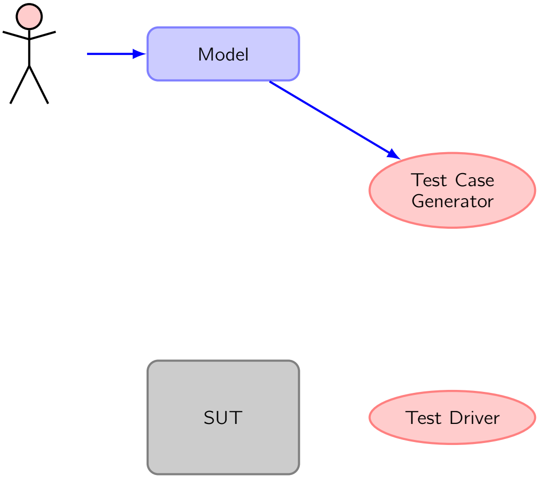
Agenda

- Mutation Testing
- Model-based Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
- Tool Demo and Examples

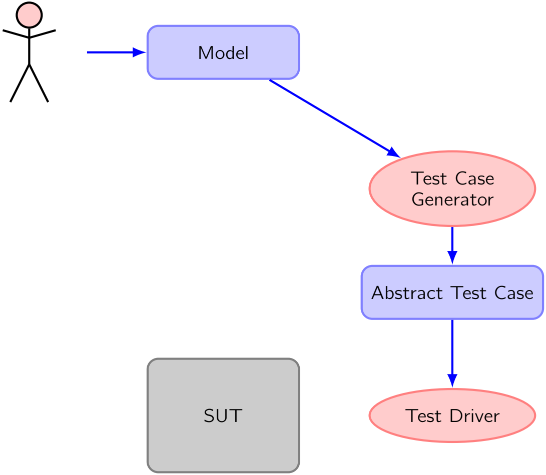
Model-Based Testing



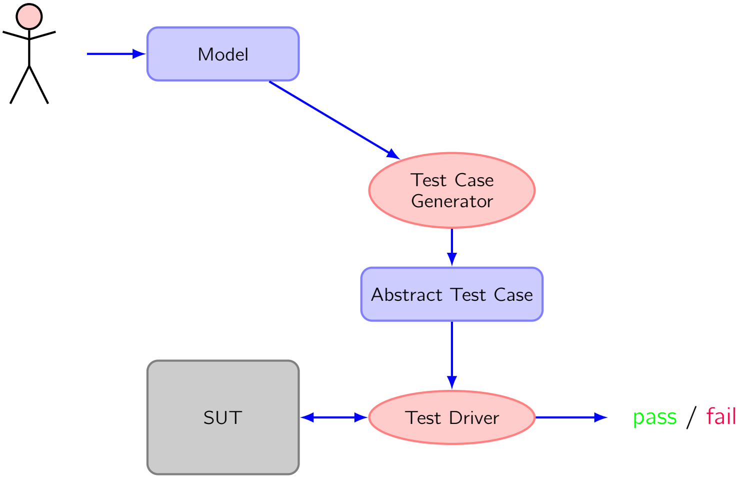
Model-Based Testing



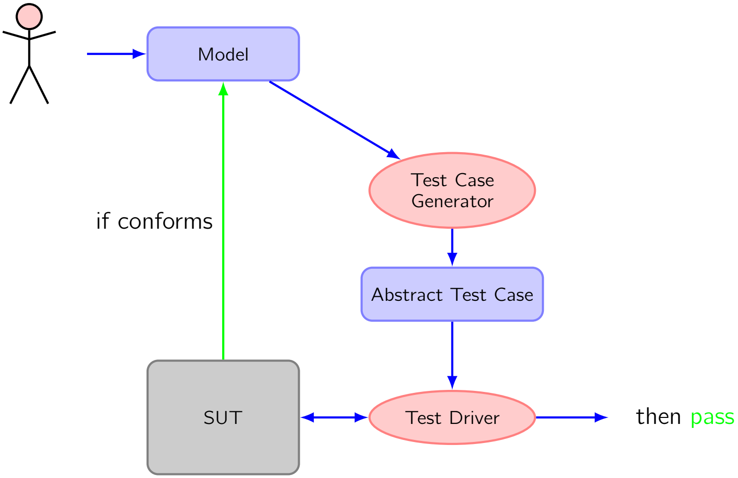
Model-Based Testing



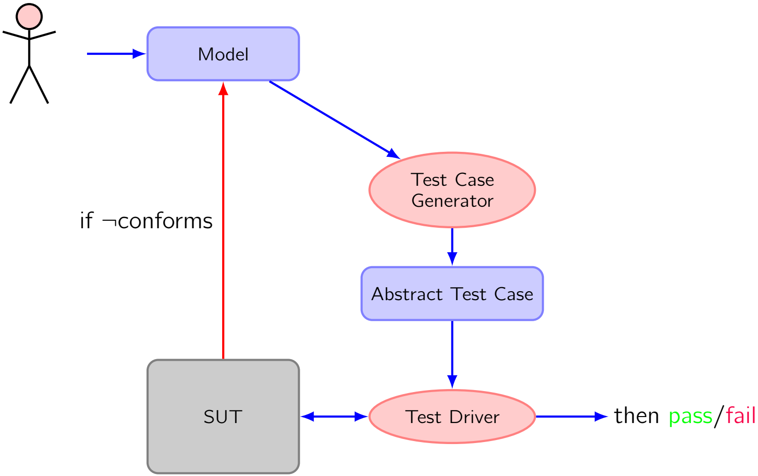
Model-Based Testing



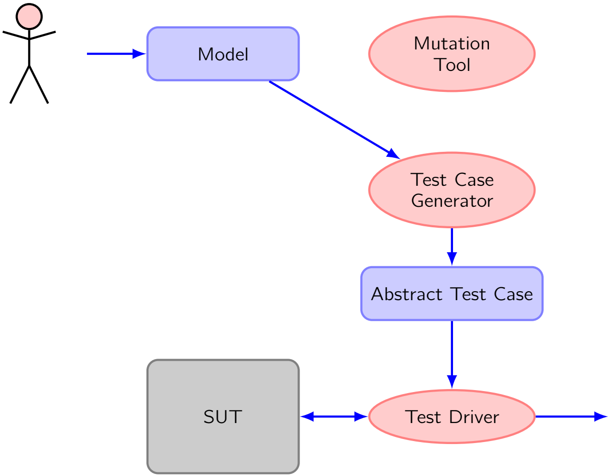
Model-Based Testing



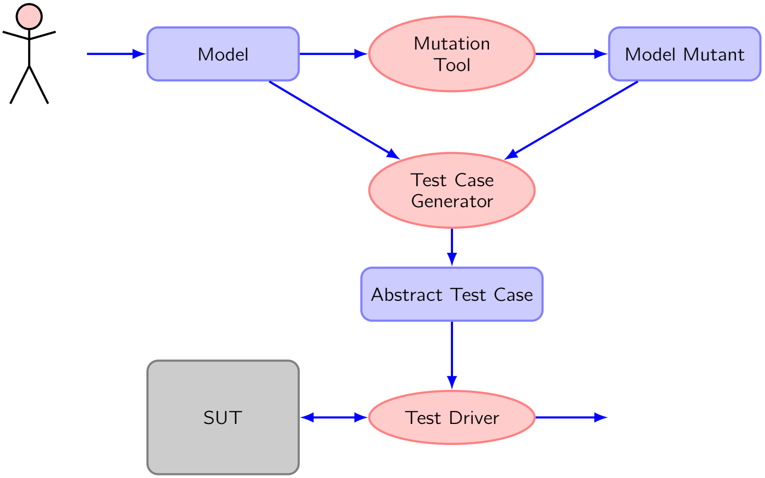
Model-Based Testing



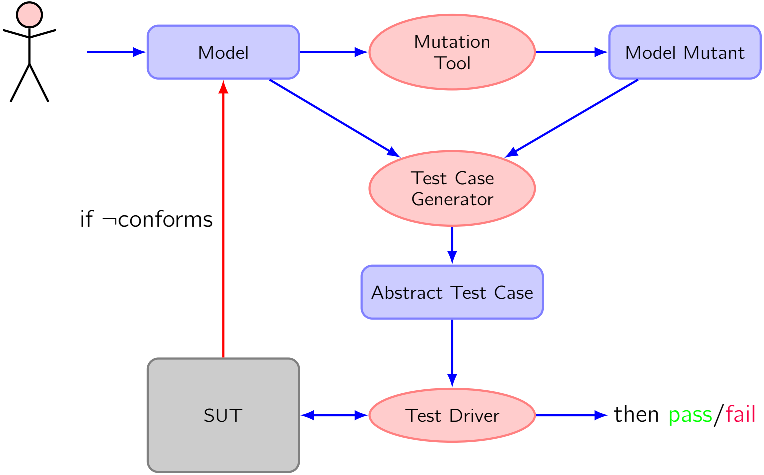
Model-Based Mutation Testing



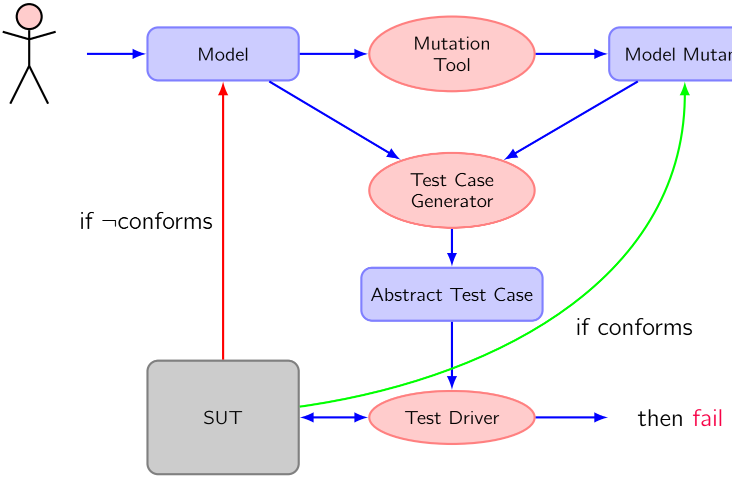
Model-Based Mutation Testing



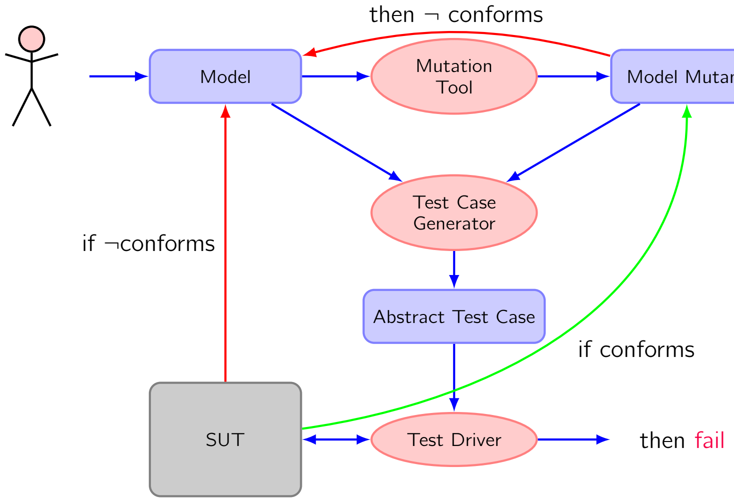
Model-Based Mutation Testing



Model-Based Mutation Testing



Model-Based Mutation Testing



Non-Conformance & Test Cases

Theorem

Given a transitive conformance relation $\sqsubseteq$, then

$$(Model \not\sqsubseteq SUT) \land (Mutant \sqsubseteq SUT) \Rightarrow (Model \not\sqsubseteq Mutant)$$

- What are the cases of non-conformance?
- Test these cases on the SUT!
- These test cases will detect if mutant has been implemented.

Test Cases as Partial Specifications

- A test case can be interpreted as a partial specification (model)
  - defines output for one input case, rest undefined.
- If a SUT (always) passes a test case, we have conformance:

$$Test\ case \sqsubseteq SUT$$

- If we generate a test case from a model, we have selected a partial behaviour such that

$$Test\ case \sqsubseteq Model$$

- If SUT conforms to the model:

$$Test\ case \sqsubseteq Model \sqsubseteq SUT$$

Test Cases as Partial Specifications

- A test case can be interpreted as a partial specification (model)
  - defines output for one input case, rest undefined.
- If a SUT (always) passes a test case, we have conformance:

$$\mathit{Test\ case} \sqsubseteq \mathit{SUT}$$

- If we generate a test case from a model, we have selected a partial behaviour such that

$$\mathit{Test\ case} \sqsubseteq \mathit{Model}$$

- If SUT conforms to the model:

$$\mathit{Test\ case} \sqsubseteq \mathit{Model} \sqsubseteq \mathit{SUT}$$

Test Cases as Partial Specifications

- A test case can be interpreted as a partial specification (model)
  - defines output for one input case, rest undefined.
- If a SUT (always) passes a test case, we have conformance:

$$\mathit{Test\ case} \sqsubseteq \mathit{SUT}$$

- If we generate a test case from a model, we have selected a partial behaviour such that

$$\mathit{Test\ case} \sqsubseteq \mathit{Model}$$

- If SUT conforms to the model:

$$\mathit{Test\ case} \sqsubseteq \mathit{Model} \sqsubseteq \mathit{SUT}$$

Test Cases as Partial Specifications

- A test case can be interpreted as a partial specification (model)
  - defines output for one input case, rest undefined.
- If a SUT (always) passes a test case, we have conformance:

$$\textit{Test case} \sqsubseteq \textit{SUT}$$

- If we generate a test case from a model, we have selected a partial behaviour such that

$$\textit{Test case} \sqsubseteq \textit{Model}$$

- If SUT conforms to the model:

$$\textit{Test case} \sqsubseteq \textit{Model} \sqsubseteq \textit{SUT}$$

Fault-Detecting Test Case

- Generated from the model
- Kills the mutant

$$\textit{Test case} \sqsubseteq \textit{Model}$$

- It is a counter-example to conformance, hence

$$\textit{Model} \not\sqsubseteq \textit{Mutant}$$

iff

$$\exists\, \textit{Test case} : (\textit{Test case} \sqsubseteq \textit{Model} \wedge \textit{Test case} \not\sqsubseteq \textit{Mutant})$$

Bernhard K. Aichernig. *Mutation Testing in the Refinement Calculus. Formal Aspects of Computing*, 15(2-3):280-295, 2003.

Fault-Detecting Test Case

- Generated from the model
- Kills the mutant

$$\mathit{Test\ case} \sqsubseteq \mathit{Model} \wedge \mathit{Test\ case} \not\sqsubseteq \mathit{Mutant}$$

- It is a counter-example to conformance, hence

$$\mathit{Model} \not\sqsubseteq \mathit{Mutant}$$

iff

$$\exists\, \mathit{Test\ case} : (\mathit{Test\ case} \sqsubseteq \mathit{Model} \wedge \mathit{Test\ case} \not\sqsubseteq \mathit{Mutant})$$

Bernhard K. Aichernig. *Mutation Testing in the Refinement Calculus. Formal Aspects of Computing*, 15(2-3):280-295, 2003.

Fault-Detecting Test Case

- Generated from the model
- Kills the mutant

$$\mathit{Test\ case} \sqsubseteq \mathit{Model} \wedge \mathit{Test\ case} \not\sqsubseteq \mathit{Mutant}$$

- It is a counter-example to conformance, hence

$$\mathit{Model} \not\sqsubseteq \mathit{Mutant}$$

iff

$$\exists\, \mathit{Test\ case} : (\mathit{Test\ case} \sqsubseteq \mathit{Model} \wedge \mathit{Test\ case} \not\sqsubseteq \mathit{Mutant})$$

Bernhard K. Aichernig. *Mutation Testing in the Refinement Calculus. Formal Aspects of Computing*, 15(2-3):280-295, 2003.

Agenda

- Mutation Testing
- Model-based Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
- Tool Demo and Examples

Transformational Systems: Semantics

- Model and Mutant interpreted as predicates $Model(s, s')$ and $Mutant(s, s')$ describing state transformations $(s \rightarrow s')$
- Conformance:

$$Model \sqsubseteq Mutant =_{df} \forall s, s' : Mutant(s, s') \Rightarrow Model(s, s')$$

- Non-conformance:

$$Model \not\sqsubseteq Mutant \;=\; \exists s, s' : Mutant(s, s') \wedge \neg Model(s, s')$$

- Read: a behaviour allowed by mutant but not by original model?
- This is a constraint satisfaction problem!

Bernhard K. Aichernig and Jifeng He. *Mutation testing in UTP*. Formal Aspects of Computing, 21(1-2):33–64, 2009.

Transformational Systems: Semantics

- Model and Mutant interpreted as predicates $Model(s, s')$ and $Mutant(s, s')$ describing state transformations ($s \rightarrow s'$)
- Conformance:

$$Model \sqsubseteq Mutant =_{df} \forall s, s' : Mutant(s, s') \Rightarrow Model(s, s')$$

- Non-conformance:

$$Model \not\sqsubseteq Mutant \;=\; \exists s, s' : Mutant(s, s') \wedge \neg Model(s, s')$$

- Read: a behaviour allowed by mutant but not by original model?
- This is a constraint satisfaction problem!

Bernhard K. Aichernig and Jifeng He. *Mutation testing in UTP*. Formal Aspects of Computing, 21(1-2):33–64, 2009.

Transformational Systems: Semantics

- Model and Mutant interpreted as predicates $Model(s, s')$ and $Mutant(s, s')$ describing state transformations $(s \to s')$
- Conformance:

$$Model \sqsubseteq Mutant =_{df} \forall s, s' : Mutant(s, s') \Rightarrow Model(s, s')$$

- Non-conformance:

$$Model \not\sqsubseteq Mutant \;=\; \exists s, s' : Mutant(s, s') \wedge \neg Model(s, s')$$

- Read: a behaviour allowed by mutant but not by original model?
- This is a constraint satisfaction problem!

Bernhard K. Aichernig and Jifeng He. *Mutation testing in UTP*. Formal Aspects of Computing, 21(1-2):33–64, 2009.

Transformational Systems: Example

Triangle semantics:

$$Mutant(a, b, c, res') \wedge \neg Model(a, b, c, res') =_{df}$$
$$(\ldots$$
$$\neg(a \leq c - b \vee a \leq b - c \vee b \leq a - c) \wedge (a \geq b \wedge b = c \wedge res' = \text{equilateral})$$
$$\ldots) \wedge$$
$$\neg(\ldots$$
$$\neg(a \leq c - b \vee a \leq b - c \vee b \leq a - c) \wedge (a = b \wedge b = c \wedge res' = \text{equilateral})$$
$$\ldots)$$

- Simplifies to $a > b \wedge b = c \wedge res' = \text{equilateral}$
- Solver produces solution: $a = 3, b = 2, c = 2, res' = \text{equilateral}$
- Test case with expected result: $a = 3, b = 2, c = 2, res' = \text{isosceles}$

Transformational Systems: Tools

Implemented with different solvers:

- OCL contracts (Constraint Handling Rules)
- Spec# contracts (Boogie, Z3)
- Reo connector language (rewriting in JTom)

Bernhard K. Aichernig and Percy Pari Salas, *Test Case Generation by OCL Mutation and Constraint Solving*, QSIC 2005.

Willibald Krenn and Bernhard K. Aichernig, *Test Case Generation by Contract Mutation in Spec#*, MBT 2009.

Sun Meng, Farhad Arbab, Bernhard K. Aichernig, Lacramioara Astefanoaei, Frank S. de Boer, and Jan Rutten. *Connectors as designs: Modeling, refinement and test case generation.* Science of Computer Programming, 77(7-8): 799-822, 2012.

Transformational Systems: Tools

Implemented with different solvers:

- OCL contracts (Constraint Handling Rules)
- Spec# contracts (Boogie, Z3)
- Reo connector language (rewriting in JTom)

Bernhard K. Aichernig and Percy Pari Salas, *Test Case Generation by OCL Mutation and Constraint Solving*, QSIC 2005.

Willibald Krenn and Bernhard K. Aichernig, *Test Case Generation by Contract Mutation in Spec#*, MBT 2009.

Sun Meng, Farhad Arbab, Bernhard K. Aichernig, Lacramioara Astefanoaei, Frank S. de Boer, and Jan Rutten. *Connectors as designs: Modeling, refinement and test case generation.* Science of Computer Programming, 77(7-8): 799-822, 2012.

Transformational Systems: Tools

Implemented with different solvers:

- OCL contracts (Constraint Handling Rules)
- Spec# contracts (Boogie, Z3)
- Reo connector language (rewriting in JTom)

Bernhard K. Aichernig and Percy Pari Salas, *Test Case Generation by OCL Mutation and Constraint Solving*, QSIC 2005.

Willibald Krenn and Bernhard K. Aichernig, *Test Case Generation by Contract Mutation in Spec#*, MBT 2009.

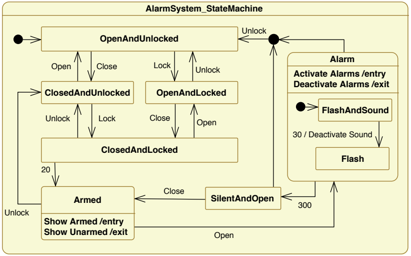
Sun Meng, Farhad Arbab, Bernhard K. Aichernig, Lacramioara Astefanoaei, Frank S. de Boer, and Jan Rutten. *Connectors as designs: Modeling, refinement and test case generation*. Science of Computer Programming, 77(7-8): 799-822, 2012.

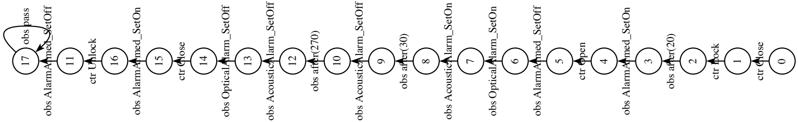
Agenda

- Mutation Testing
- Model-based Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
- Tool Demo and Examples

Reactive Systems

- React to the environment
- Do not terminate
- Servers and Controllers
- Events: controllable and observable communication events
- Test cases: sequences of events





Adaptive test cases: trees branching at non-deterministic observations

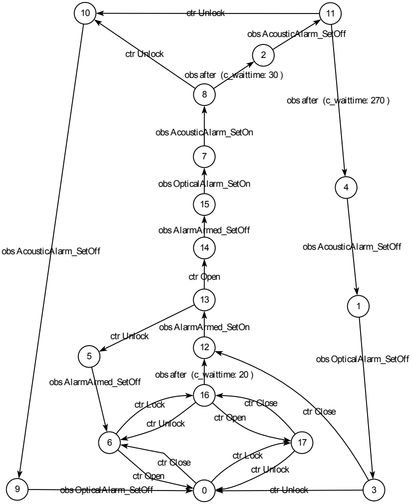
Semantics

- Operational semantics
  e.g. Labelled Transition Systems
- Input-output conformance (ioco)
  [Tretmans96]

SUT ioco $Model =_{df}$

$\forall \sigma \in traces(Model)$:
$out(SUT \text{ after } \sigma) \subseteq out(Model \text{ after } \sigma)$

out ... outputs + quiescence
after ... reachable states after trace



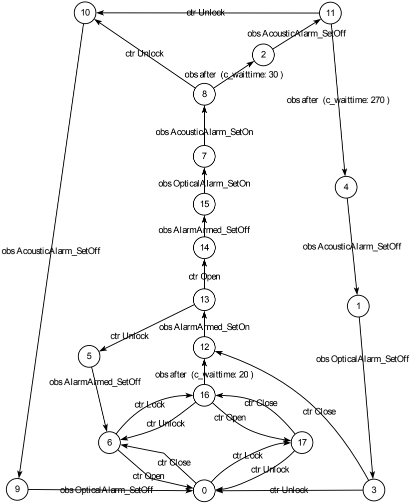
Semantics

- Operational semantics
  e.g. Labelled Transition Systems
- Input-output conformance (ioco)
  [Tretmans96]

SUT ioco $Model =_{df}$

$\forall \sigma \in$ traces($Model$) :
out(SUT after σ) $\subseteq$ out($Model$ after σ)

out … outputs + quiescence
after … reachable states after trace



Semantics

- Operational semantics
  e.g. Labelled Transition Systems
- Input-output conformance (ioco) [Tretmans96]

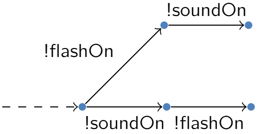
SUT ioco *Model* $=_{df}$

$\forall \sigma \in$ traces(*Model*) :

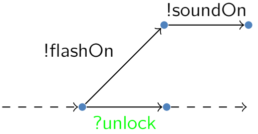
out(*SUT* after σ) $\subseteq$ out(*Model* after σ)

out ... outputs + quiescence
after ... reachable states after trace

Model:



SUT:

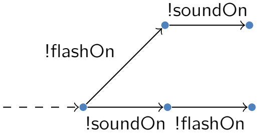


SUT ioco Model ✓

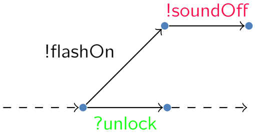
Explicit Conformance Checking

- Model and Mutant → LTS
- Determinisation

Model:



Mutant:



- Build synchronous product modulo ioco
- If mutant has additional
  - !output: → fail sink state
  - ?input: → pass sink state

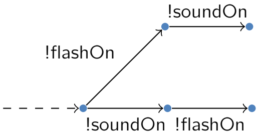
Model $\times_{ioco}$ Mutant:

- Extract test case covering fail state

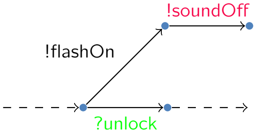
Explicit Conformance Checking

- Model and Mutant → LTS
- Determinisation

Model:

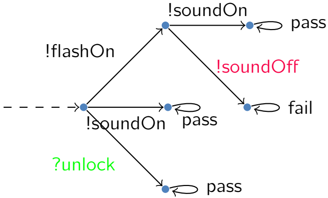


Mutant:



- Build synchronous product modulo ioco
- If mutant has additional
  - !output: → fail sink state
  - ?input: → pass sink state

Model $\times_{ioco}$ Mutant:

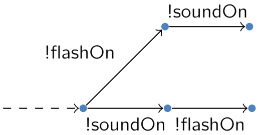


- Extract test case covering fail state

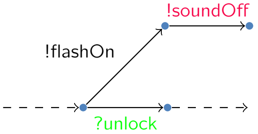
Explicit Conformance Checking

- Model and Mutant → LTS
- Determinisation

Model:

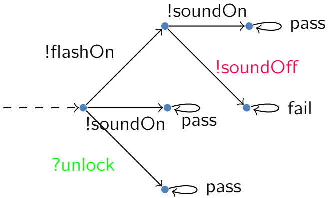


Mutant:



- Build synchronous product modulo ioco
- If mutant has additional
  - !output: → fail sink state
  - ?input: → pass sink state

Model $\times_{ioco}$ Mutant:



- Extract test case covering fail state

Applications of Explicit Conformance Checking

- HTTP Server (LOTOS)
- SIP Server (LOTOS)
- Controllers (UML)
- Hybrid Systems (Action System)

Scalability: abstractions for data-intensive models

Bernhard K. Aichernig and Corrales Delgado. *From Faults via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems*, FASE 2006.

Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. *Fault-based conformance testing in practice*. International Journal of Software and Informatics, 3(2-3):375-411, 2009. Chinese Academy of Science.

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. *Efficient mutation killers in action*, ICST 2011.

Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. *Automated conformance verification of hybrid systems*, QSIC 2010.

Applications of Explicit Conformance Checking

- HTTP Server (LOTOS)
- SIP Server (LOTOS)
- Controllers (UML)
- Hybrid Systems (Action System)

Scalability: abstractions for data-intensive models

Bernhard K. Aichernig and Corrales Delgado. *From Faults via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems*, FASE 2006.

Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. *Fault-based conformance testing in practice*. International Journal of Software and Informatics, 3(2-3):375-411, 2009. Chinese Academy of Science.

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. *Efficient mutation killers in action*, ICST 2011.

Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. *Automated conformance verification of hybrid systems*, QSIC 2010.

Applications of Explicit Conformance Checking

- HTTP Server (LOTOS)
- SIP Server (LOTOS)
- Controllers (UML)
- Hybrid Systems (Action System)

Scalability: abstractions for data-intensive models

Bernhard K. Aichernig and Corrales Delgado. *From Faults via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems*, FASE 2006.

Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. *Fault-based conformance testing in practice*. International Journal of Software and Informatics, 3(2-3):375-411, 2009. Chinese Academy of Science.

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. *Efficient mutation killers in action*, ICST 2011.

Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. *Automated conformance verification of hybrid systems*, QSIC 2010.

Applications of Explicit Conformance Checking

- HTTP Server (LOTOS)
- SIP Server (LOTOS)
- Controllers (UML)
- Hybrid Systems (Action System)

Scalability: abstractions for data-intensive models

Bernhard K. Aichernig and Corrales Delgado. *From Faults via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems*, FASE 2006.

Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. *Fault-based conformance testing in practice*. International Journal of Software and Informatics, 3(2-3):375-411, 2009. Chinese Academy of Science.

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. *Efficient mutation killers in action*, ICST 2011.

Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. *Automated conformance verification of hybrid systems*, QSIC 2010.

Applications of Explicit Conformance Checking

- HTTP Server (LOTOS)
- SIP Server (LOTOS)
- Controllers (UML)
- Hybrid Systems (Action System)

Scalability: abstractions for data-intensive models

Bernhard K. Aichernig and Corrales Delgado. *From Faults via Test Purposes to Test Cases: On the Fault-Based Testing of Concurrent Systems*, FASE 2006.

Martin Weiglhofer, Bernhard K. Aichernig, and Franz Wotawa. *Fault-based conformance testing in practice*. International Journal of Software and Informatics, 3(2-3):375-411, 2009. Chinese Academy of Science.

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, and Willibald Krenn. *Efficient mutation killers in action*, ICST 2011.

Harald Brandl, Martin Weiglhofer, and Bernhard K. Aichernig. *Automated conformance verification of hybrid systems*, QSIC 2010.

Action Systems

- Action Systems [Back83]
- Non-deterministic choice of actions
- Actions are guarded commands
- Loop over Actions
- Terminates if all guards disabled
- Actions are labelled and represent events
- Two semantics:
  - Labelled Transition Systems
  - Predicative semantics

var $closed : Bool := false;$
$locked : Bool := false;$
$armed : Bool := false;$
$sound : Bool := false;$
$flash : Bool := false;$
actions
$Close :: \neg closed \rightarrow closed := true;$

$Open :: closed \rightarrow closed := false;$

$SoundOn :: armed \wedge \neg closed \wedge \neg sound \rightarrow$
$sound := true;$

$FlashOn :: armed \wedge \neg closed \wedge \neg flash \rightarrow$
$flash := true$

...

do $Close$
$\square$
$Open$
$\square$
$SoundOn; FlashOn$
$\square$
$FlashOn; SoundOn$
...
od

Predicative Semantics of Action Systems

The transition relation (one step) is

- translated to a constraint over state variables s and event-traces tr:

$$
\begin{aligned}
l :: g \rightarrow B &=_{df} g \ \wedge \ B \ \wedge \ tr' = tr ^\frown [l] \\
l(\overline{x}) :: g \rightarrow B &=_{df} \exists \ \overline{x} : g \ \wedge \ B \ \wedge \ tr' = tr ^\frown [l(\overline{x})] \\
x := e &=_{df} x' = e \ \wedge \ y' = y \ \wedge \ldots \wedge \ z' = z \\
g \rightarrow B &=_{df} g \ \wedge \ B \\
B(s, s'); B(s, s') &=_{df} \exists \ s_0 : B(s, s_0) \ \wedge \ B(s_0, s') \\
B_1 \ \Box \ B_2 &=_{df} B_1 \ \vee \ B_2
\end{aligned}
$$

- then simplified (DNF + quantifier elimination)

Symbolic Conformance Checking

$$\exists\ s, s', tr, tr' : reachable(s, tr) \ \wedge\ Mutant(s, s', tr, tr') \ \wedge\ \neg Model(s, s', tr, tr')$$

- Is non-conformance reachable?
- Fast, but stronger than ioco.
- Ioco for complete models:

$$\exists\ s_1, s_1', s_2, s_2', tr, !a : reachable(Mutant, tr, s_1) \ \wedge\ reachable(Model, tr, s_2)$$

$$\wedge$$

$$Mutant(s1, s1', tr, tr \frown !a) \ \wedge\ \neg Model(s2, s2', tr, tr \frown !a)$$

Symbolic Conformance Checking

$$\exists\ s, s', tr, tr' : reachable(s, tr) \wedge Mutant(s, s', tr, tr') \wedge \neg Model(s, s', tr, tr')$$

- Is non-conformance reachable?
- Fast, but stronger than ioco.
- Ioco for complete models:

$$\exists\ s_1, s_1', s_2, s_2', tr, !a : reachable(Mutant, tr, s_1) \wedge reachable(Model, tr, s_2)$$

$$\wedge$$

$$Mutant(s1, s1', tr, tr ^\frown !a) \wedge \neg Model(s2, s2', tr, tr ^\frown !a)$$

Symbolic Conformance Checkers

- Two implementations for Action Systems
  - Constraint Logic Programming: Sicstus Prolog
  - SMT solving: Scala + Z3
- Timed Automata: Scala + Z3 (tioco)
- After optimisations:

Bernhard K. Aichernig and Elisabeth Jöbstl. *Towards symbolic model-based mutation testing: Combining reachability and refinement checking*, MBT 2012.

Bernhard K. Aichernig and Elisabeth Jöbstl. *Efficient Refinement Checking for Model-Based Mutation Testing*, QSIC 2012.

Bernhard K. Aichernig, Florian Lorber and Dejan Nickovic. *Time for Mutants: Mutation testing with timed automata*, TAP 2013

Bernhard K. Aichernig, Elisabeth Jöbstl and Matthias Kegele. *Incremental refinement checking for test case generation*, TAP 2013

Symbolic Conformance Checkers

- Two implementations for Action Systems
  - Constraint Logic Programming: Sicstus Prolog
  - SMT solving: Scala + Z3
- Timed Automata: Scala + Z3 (tioco)
- After optimisations:

Bernhard K. Aichernig and Elisabeth Jöbstl. *Towards symbolic model-based mutation testing: Combining reachability and refinement checking*, MBT 2012.

Bernhard K. Aichernig and Elisabeth Jöbstl. *Efficient Refinement Checking for Model-Based Mutation Testing*, QSIC 2012.

Bernhard K. Aichernig, Florian Lorber and Dejan Nickovic. *Time for Mutants: Mutation testing with timed automata*, TAP 2013

Bernhard K. Aichernig, Elisabeth Jöbstl and Matthias Kegele. *Incremental refinement checking for test case generation*, TAP 2013

Symbolic Conformance Checkers

- Two implementations for Action Systems
  - Constraint Logic Programming: Sicstus Prolog
  - SMT solving: Scala + Z3
- Timed Automata: Scala + Z3 (tioco)
- After optimisations:

Prolog and SMT equally fast!

Bernhard K. Aichernig and Elisabeth Jöbstl. *Towards symbolic model-based mutation testing: Combining reachability and refinement checking*, MBT 2012.

Bernhard K. Aichernig and Elisabeth Jöbstl. *Efficient Refinement Checking for Model-Based Mutation Testing*, QSIC 2012.

Bernhard K. Aichernig, Florian Lorber and Dejan Nickovic. *Time for Mutants: Mutation testing with timed automata*, TAP 2013

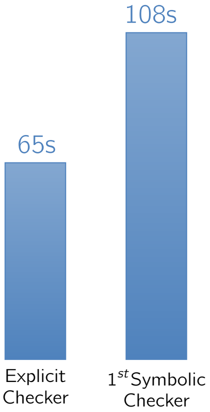
Bernhard K. Aichernig, Elisabeth Jöbstl and Matthias Kegele. *Incremental refinement checking for test case generation*, TAP 2013

Optimisations

Performance gains for checking 207 mutants of the Car Alarm System.

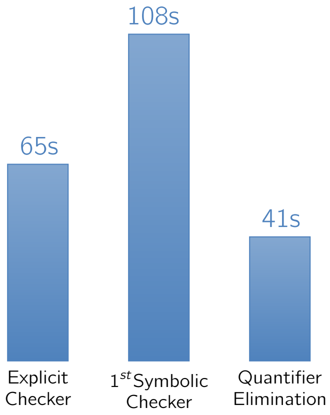
Optimisations

Performance gains for checking 207 mutants of the Car Alarm System.



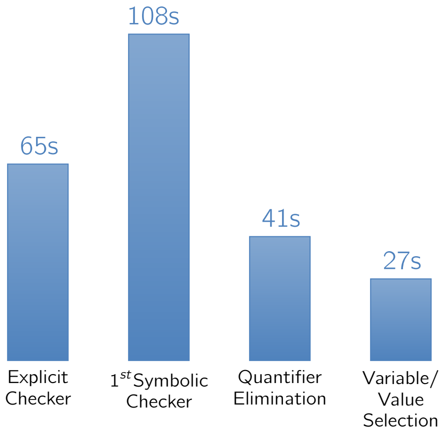
Optimisations

Performance gains for checking 207 mutants of the Car Alarm System.



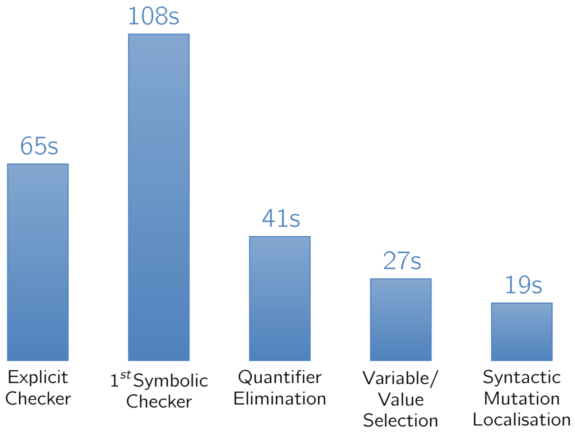
Optimisations

Performance gains for checking 207 mutants of the Car Alarm System.



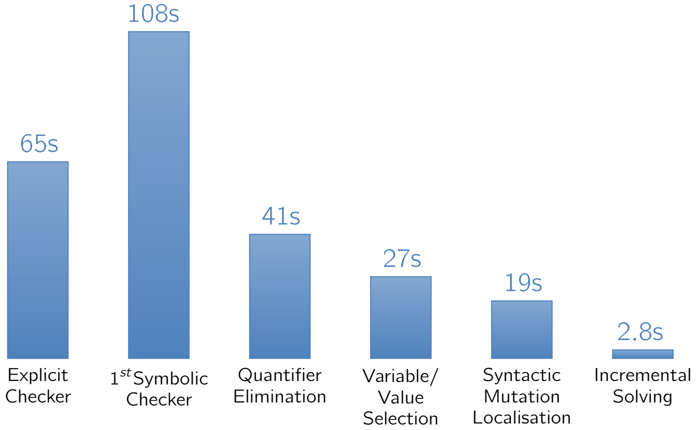
Optimisations

Performance gains for checking 207 mutants of the Car Alarm System.



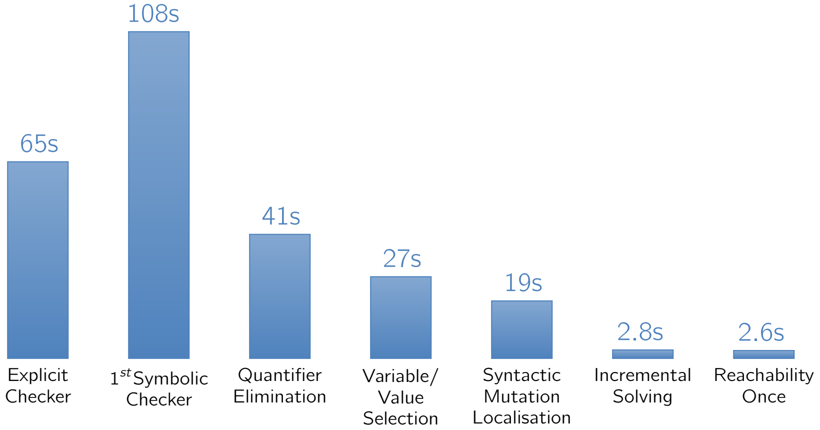
Optimisations

Performance gains for checking 207 mutants of the Car Alarm System.



Optimisations

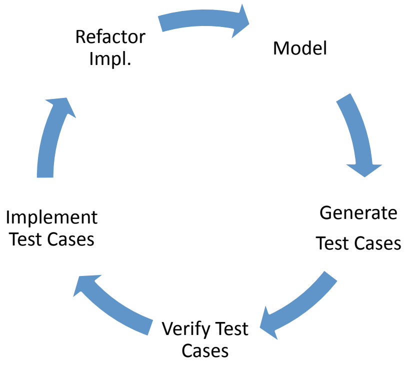
Performance gains for checking 207 mutants of the Car Alarm System.



Agenda

- Mutation Testing
- Model-based Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
- Tool Demo and Examples

Agile Development



- Model-driven development
- Model-based test case generation
- Formal verification
- Test-driven development

Agenda

- Mutation Testing
- Model-based Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
- Tool Demo and Examples

MoMuT Tools

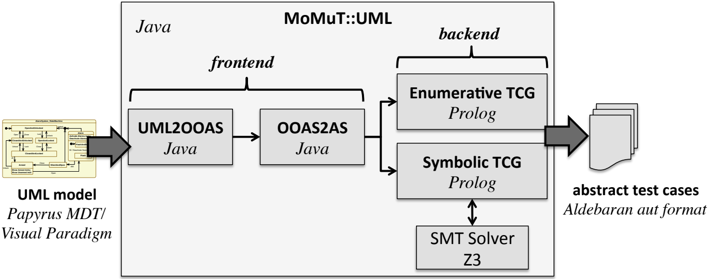
MoMuT

- is a family of tools implementing Model-based Mutation Testing.
- is jointly developed and maintained by AIT and TU Graz
- supports different modelling styles:
  - MoMuT::UML
  - MoMuT::OOAS
  - MoMuT::TA
  - MoMuT::Reqs

www.momut.org

MoMuT::UML

- Test-case generator of AIT and TU Graz
- Implementing model-based mutation testing for UML state machines

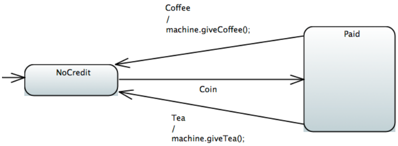


Architecture of the *MoMuT::UML* tool chain

AS ... Action Systems [Back83]

OOAS ... Object-Oriented Action Systems

MoMuT::UML



- Enumerative back-end: ioco
- Symbolic back-end supports two conformance relations:
  - State-based Refinement
  - Event-based ioco

Bernhard Aichernig, Harald Brandl, Elisabeth Jobstl, Willibald Krenn, Rupert Schlick and Stefan Tiran. *MoMuT::UML Model-Based Mutation Testing for UML*, ICST 2015.

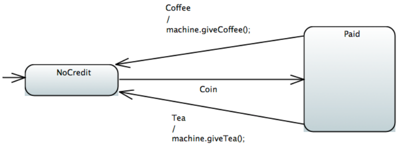
Bernhard K. Aichernig, Jakob Auer, Elisabeth Jobstl, Robert Korosec, Willibald Krenn, Rupert Schlick and Birgit Vera Schmidt. *Model-Based Mutation Testing of an Industrial Measurement Device*, TAP 2014.

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jobstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. *Killing strategies for model-based mutation testing*, Software Testing, Verification and Reliability, 2014.

Combined conformance checking:

- Refinement checker searches for faulty state (fast)
- Ioco checker looks if faulty state propagates to different observations

MoMuT::UML



- Enumerative back-end: ioco
- Symbolic back-end supports two conformance relations:
  - State-based Refinement
  - Event-based ioco

Bernhard Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick and Stefan Tiran. *MoMut: : UML Model-Based Mutation Testing for UML*, ICST 2015.

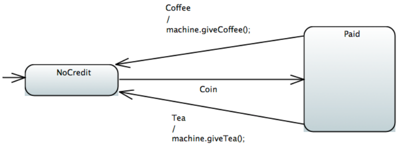
Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick and Birgit Vera Schmidt. *Model-Based Mutation Testing of an Industrial Measurement Device*, TAP 2014.

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. *Killing strategies for model-based mutation testing*, Software Testing, Verification and Reliability, 2014

Combined conformance checking:

- Refinement checker searches for faulty state (fast)
- Ioco checker looks if faulty state propagates to different observations

MoMuT::UML



- Enumerative back-end: ioco
- Symbolic back-end supports two conformance relations:
  - State-based Refinement
  - Event-based ioco

Bernhard Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick and Stefan Tiran. *MoMut: : UML Model-Based Mutation Testing for UML*, ICST 2015.

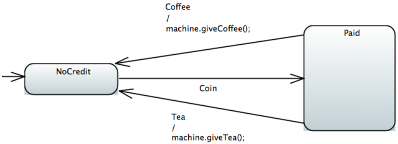
Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick and Birgit Vera Schmidt. *Model-Based Mutation Testing of an Industrial Measurement Device*, TAP 2014.

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. *Killing strategies for model-based mutation testing*, Software Testing, Verification and Reliability, 2014

Combined conformance checking:

- Refinement checker searches for faulty state (fast)
- Ioco checker looks if faulty state propagates to different observations

MoMuT::UML



- Enumerative back-end: ioco
- Symbolic back-end supports two conformance relations:
  - State-based Refinement
  - Event-based ioco

Combined conformance checking:

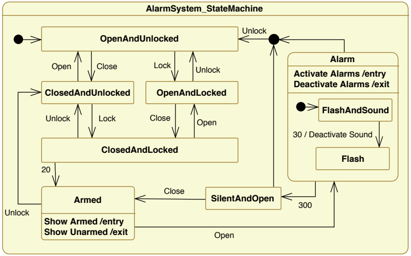
- Refinement checker searches for faulty state (fast)
- Ioco checker looks if faulty state propagates to different observations

Bernhard Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick and Stefan Tiran. *MoMut: : UML Model-Based Mutation Testing for UML*, ICST 2015.

Bernhard K. Aichernig, Jakob Auer, Elisabeth Jöbstl, Robert Korosec, Willibald Krenn, Rupert Schlick and Birgit Vera Schmidt. *Model-Based Mutation Testing of an Industrial Measurement Device*, TAP 2014.

Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. *Killing strategies for model-based mutation testing*, Software Testing, Verification and Reliability, 2014

Case Study 1: Car Alarm System

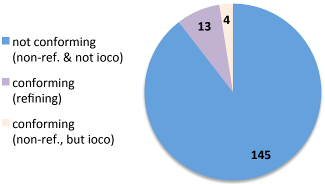


State machine model in UML

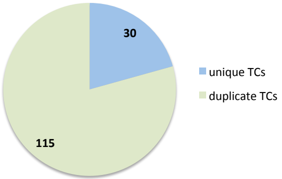
	CAS_UML
actions [#]	51
state variables [#]	35
possible states [#]	$1.7 \cdot 10^{18}$
reachable states [#]	229
required exploration depth	17

Metrics of Generated Action System

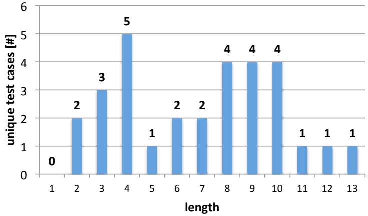
Case Study 1: TCG



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Case Study 1: Fault Propagation

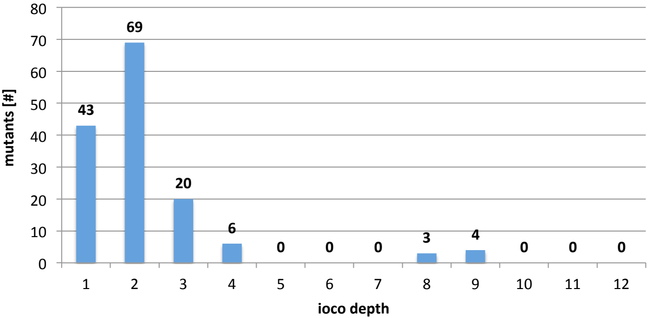


Figure: Number of steps from fault to failure (ioco depths)

Case Study 1: Run-times

... for combined conformance checking (in sec., max. depth 20+20) :

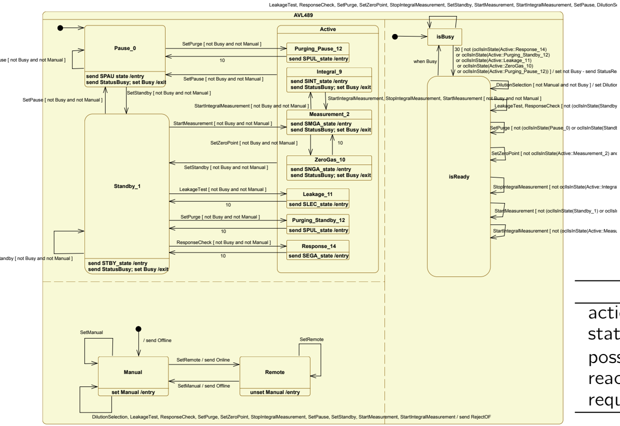
		conforming (refining)	conforming (non-ref., but ioco)	not conforming (non-ref. & not ioco)	total
mutants [#]		13	4	145	162
ref. check	Σ	4.03	1.63	56.41	62.07
	ϕ	0.31	0.41	0.39	0.38
	max	0.41	0.44	0.53	0.53
ioco check	Σ	-	17.71	1.9 min	2.2 min
	ϕ	-	4.43	0.79	0.81
	max	-	4.48	2.01	4.48
tc constr.	Σ	-	-	1.3 min	1.3 min
	ϕ	-	-	0.55	0.49
	max	-	-	1.48	1.48
total without logging	Σ	4.25	19.4	4.2 min	**4.6 min**
	ϕ	0.33	4.85	1.74	1.7
	max	0.43	4.89	2.77	4.89

Comparison to stand-alone ioco-check with depth 20: 5.1 min

Case Study 2: AVL489 Particle Counter

- One of AVL's automotive measurement devices
- Measures particle number concentrations in exhaust gas
- Focus: testing of the control logic

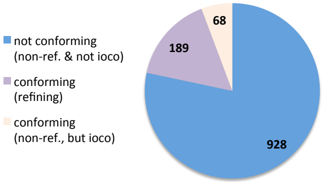
Case Study 2: Test Model of AVL489



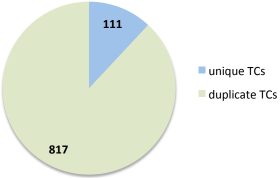
	PC_UML
actions [#]	109
state variables [#]	74
possible states [#]	$1.2 \cdot 10^{31}$
reachable states [#]	$> 850\,700$
required exploration depth	> 25

Metrics of Generated Action System

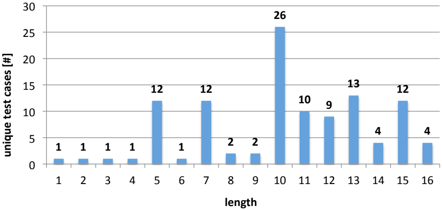
Case Study 2: TCG



(a) Breakup into conforming and not conforming model mutants.



(b) Breakup into unique and duplicate test cases.



(c) Lengths of the unique test cases.

Case Study 2: Fault Propagation

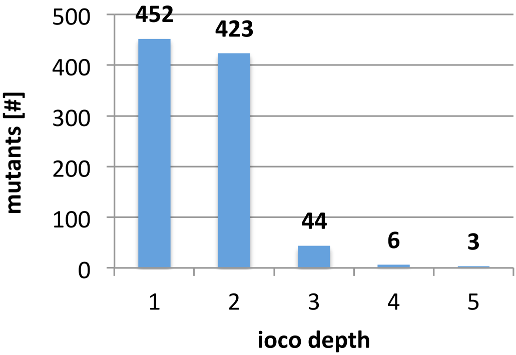


Figure: Number of steps from fault to failure (ioco depths)

Case Study 2: Run-times

... for combined conformance checking (in min., max. depth 15+5) :

		conforming (refining)	conforming (non-ref., but ioco)	not conforming (non-ref. & not ioco)	total
mutants [#]		189	68	928	1185
ref. check	Σ	6.1 h	7.7	7.1 h	13.3 h
	ϕ	1.9	6.8 sec	27 sec	40 sec
	max	4.3	1.8	3.9	4.3
ioco check	Σ	-	0.7 h	1.7 h	2.4 h
	ϕ	-	38 sec	7 sec	7.4 sec
	max	-	2	27 sec	2
tc constr.	Σ	-	-	22.9	22.9
	ϕ	-	-	1.5 sec	1.2 sec
	max	-	-	3.7 sec	3.7 sec
total without logging	Σ	6.1 h	0.9 h	9.2 h	**16.2 h**
	ϕ	1.9	0.8	0.6	0.8
	max	4.3	2.2	4.1	4.3

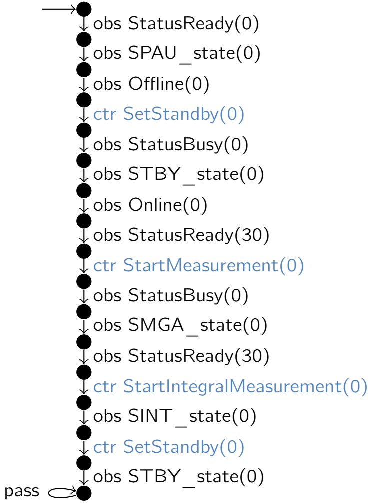
Case Study 2: Run-times

... comparison to stand-alone ioco check (in min., max. depth 10):

		not ioco	ioco	total
mutants [#]		719	466	1185
time – ioco check	Σ	9.8 h	22.8 h	32.6 h
	ϕ	0.8	2.9	1.7
	max	3.9	5.2	5.2
time – tc constr.	Σ	19	-	19
	ϕ	1.6 sec	-	1 sec
	max	5.8 sec	-	5.8 sec
total without logging	Σ	10.1 h	22.8 h	**32.9 h**
	ϕ	0.8	2.9	1.7
	max	3.9	5.2	5.2

appr. 16h vs. 33h

Abstract Test Case of AVL489



Abstract test cases → concrete C# NUnit test cases.

ctr … controllable event (input)

obs … observable event (output)

Test Execution on Particle Counter

We found several bugs in the SUT:

- Forbidden changes of operating state while busy
  - Pause → Standby
  - Normal Measurement → Integral Measurement
- Ignoring high-frequent input without error-messages
- Loss of error messages in client for remote control of the device

MoMuT::UML Reimplementation

Motivation: Railway Interlocking System (Thales)

- Reimplementation of enumerative TCG in C by AIT
- Assuming deterministic systems
- ioco checking $\Rightarrow$ ioco testing (random)
- Short lived mutants: create mutants while exploring

MoMuT::OOAS

Object-Oriented Action Systems:

- Textual model programs
- Guarded Actions in do-od loop
- Modularization via objects
- Communication via methods
- Mutation directly on OOAS

Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. *Mapping UML to labeled transition systems for test-case generation - a translation via object-oriented action systems*. FMCO, 2009

```
types
    CoffeeMachine = autocons system
|[  var
        paid : Boolean = false ;
        coffee_sel : Boolean = false
    actions
        ctr coin =
            requires true :
                paid := true
        end;
        ctr coffeebutton =
            requires paid :
                coffee_sel := true;
                paid := false ;
        end ;
        obs coffee =
            requires coffee_sel :
                skip
        end ;
do
    coin () [] coffeebutton () [] coffee ()
od
]| system CoffeeMachine
```

MoMuT::OOAS

Object-Oriented Action Systems:

- Textual model programs
- Guarded Actions in do-od loop
- Modularization via objects
- Communication via methods
- Mutation directly on OOAS

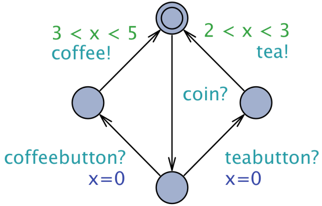
Willibald Krenn, Rupert Schlick, and Bernhard K. Aichernig. *Mapping UML to labeled transition systems for test-case generation - a translation via object-oriented action systems*, FMCO, 2009

```
types
    CoffeeMachine = autocons system
|[  var
        paid  :  Boolean = false  ;
        coffee_sel  :  Boolean = false
    actions
        ctr  coin  =
            requires  true  :
                paid  :=  true
        end;
        ctr  coffeebutton  =
            requires  paid  :
                coffee_sel  :=  true ;
                paid  :=  false ;
        end  ;
        obs  coffee  =
            requires  coffee_sel  :
                skip
        end  ;
do
    coin ()  []  coffeebutton ()  []  coffee ()
od
]|  system  CoffeeMachine
```

MoMuT::TA

Timed Automata:

- Modelling in UPPAAL model checker
- Finite-state machines with real-valued clock variables
- Time passage in locations
- Time restrictions on locations and guards



MoMuT::TA (cont.)

- tioco-conformance: *M tioco S*
  - $out(M) \subseteq out(S)$
  - time delay is an output
- Conformance check via language inclusion
  - Requires deterministic automata
  - SMT-Solver Z3
- Determinization

Application: Crystal Usecase (Volvo)

Bernhard K. Aichernig, Florian Lorber and Dejan Nickovic. *Time for Mutants: Mutation testing with timed automata*, TAP 2013

Bernhard K. Aichernig and Florian Lorber. *Towards generation of adaptive test cases from partial models of determinized timed automata*, A-MOST 2014.

Florian Lorber, Amnon Rosenmann, Dejan Nickovic and Bernhard K. Aichernig. *Bounded Determinization of Timed Automata with Silent Transitions*, FORMATS 2015?

MoMuT::TA (cont.)

- tioco-conformance: $M\ tioco\ S$
  - $out(M) \subseteq out(S)$
  - time delay is an output
- Conformance check via language inclusion
  - Requires deterministic automata
  - SMT-Solver Z3
- Determinization

Application: Crystal Usecase (Volvo)

Bernhard K. Aichernig, Florian Lorber and Dejan Nickovic. *Time for Mutants: Mutation testing with timed automata*, TAP 2013

Bernhard K. Aichernig and Florian Lorber. *Towards generation of adaptive test cases from partial models of determinized timed automata*, A-MOST 2014.

Florian Lorber, Amnon Rosenmann, Dejan Nickovic and Bernhard K. Aichernig. *Bounded Determinization of Timed Automata with Silent Transitions*, FORMATS 2015?

MoMuT::REQs

Contract-based Requirement Interfaces:

- Synchronous assume-guarantee pairs
- Combined via conjunction
- No model-based mutation testing yet

Application: Airbag Chip (Infineon)

```
Inputs  coin, teabutton, coffeebutton;
Outputs coffee, tea;
 Internals  paid;

{I}  not paid and not coffee and not tea
{R1} assume coin'
     guarantee paid'
{R2} assume paid and teabutton' and not coffeebutton'
     guarantee tea' and not paid'
{R3} assume paid and coffeebutton' and not teabutton'
     guarantee coffee' and not paid'

{R4} assume teabutton' and coffeebutton'
     guarantee skip
```

Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, Stefan Tiran. *Require, Test and Trace IT*, FMICS 2015

Bernhard K. Aichernig and Dejan Nickovic and Stefan Tiran. *Scalable Incremental Test-case Generation from Large Behavior Models*, TAP 2015.

Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, Rupert Schlick, Didier Simoneau, Stefan Tiran. *Integration of Requirements Engineering and Test-Case Generation via OSLC*, QSIC 2014

MoMuT::REQs

Contract-based Requirement Interfaces:

- Synchronous assume-guarantee pairs
- Combined via conjunction
- No model-based mutation testing yet

Application: Airbag Chip (Infineon)

```
Inputs   coin,  teabutton,  coffeebutton;
Outputs  coffee,  tea;
 Internals  paid;

{I}  not paid and not coffee and not tea
{R1} assume coin'
     guarantee paid'
{R2} assume paid and teabutton' and not coffeebutton'
     guarantee tea' and not paid'
{R3} assume paid and coffeebutton' and not teabutton'
     guarantee coffee' and not paid'

{R4} assume teabutton' and coffeebutton'
     guarantee skip
```

Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, Stefan Tiran. *Require, Test and Trace IT*, FMICS 2015

Bernhard K. Aichernig and Dejan Nickovic and Stefan Tiran. *Scalable Incremental Test-case Generation from Large Behavior Models*, TAP 2015.

Bernhard K. Aichernig, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, Rupert Schlick, Didier Simoneau, Stefan Tiran. *Integration of Requirements Engineering and Test-Case Generation via OSLC*, QSIC 2014

Agenda

- Mutation Testing
- Model-based Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
- Tool Demo and Examples

Tool Demo

Conclusions

- Model-based Testing + Mutation Testing
- Formal semantics $\rightarrow$ test case generators $\rightarrow$ industry
- Novelty: general theory + tools for non-deterministic models + different modelling styles
- Future:
  - domain-specific models
  - non-functional fault models (resource limitations)

Testing cannot show the absence of bugs [Dijkstra72].

Testing can show the absence of specific bugs [Aichernig15].

Conclusions

- Model-based Testing + Mutation Testing
- Formal semantics → test case generators → industry
- Novelty: general theory + tools for non-deterministic models + different modelling styles
- Future:
  - domain-specific models
  - non-functional fault models (resource limitations)

Testing cannot show the absence of bugs [Dijkstra72].

Testing can show the absence of specific bugs [Aichernig15].

Conclusions

- Model-based Testing + Mutation Testing
- Formal semantics → test case generators → industry
- Novelty: general theory + tools for non-deterministic models + different modelling styles
- Future:
  - domain-specific models
  - non-functional fault models (resource limitations)

Testing cannot show the absence of bugs [Dijkstra72].

Testing can show the absence of specific bugs [Aichernig15].