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

- **Formal Methods**
  - Precise abstraction
  - Refinement
  - Compositionality

- **Falsification**
  - Model-based testing
  - Model learning

- **Verification**
  - Proof-based development
  - Statistical model checking

- **Specification & Design**

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
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

First actual case of bug being found.
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 :(

```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 :( 

```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]

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]

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
  - 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)`

```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"
    }
}
```
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)`

```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"
    }
  }
}
```
Example: Reactive System

- Mutant survives
  - function coverage
  - state coverage
  - transition coverage

- Killed by
  - Lock();
  - Close();
  - After(20);
Example: Reactive System

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

\[\langle \neg\text{setEmptyOn}, \?\text{Enqueue}, \neg\text{setEmptyOff}, \?\text{Enqueue}, \?\text{Enqueue}, \?\text{Enqueue}, \?\text{Enqueue}, \?\text{Dequeue}, \neg\text{setFullOn}, \?\text{Enqueue}, \neg\text{setFullOff}, \?\text{Enqueue}, \neg\text{setFullOn} \rangle\]
From Analysis to Synthesis

State of art:

**Analysis of test cases**

How many mutants killed by test cases?

\[
\text{mutation score} = \frac{\#\text{killed mutants}}{\#\text{mutants}}
\]

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?

\[ \text{mutation score} = \frac{\# \text{killed mutants}}{\# \text{mutants}} \]

**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?

\[ \text{mutation score} = \frac{\# \text{killed mutants}}{\# \text{mutants}} \]

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?

\[ \text{mutation score} = \frac{\# \text{killed mutants}}{\# \text{mutants}} \]

**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 (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

- Test cases
  - Test execution
    - SUT
      - pass
      - fail
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

- High-level test notation
- Test execution
- SUT
- Pass
- Fail
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

- Test case generation
- Test execution
- System model
- SUT
- Conformance
- All tests pass
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
- assignment of verdicts
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

- SUT
- Test Case Generator
- Test Driver
Model-Based Testing

- Model
- Test Case Generator
- SUT
- Test Driver
Model-Based Testing

- Model
- Test Case Generator
- Abstract Test Case
- Test Driver
- SUT

B.K. Aichernig HSST 2018 Model-based Mutation Testing
Model-Based Testing

- Model
- Test Case Generator
- Abstract Test Case
- SUT
- Test Driver
- pass / fail
Model-Based Testing

if conforms

then pass
Model-Based Testing

If $\neg$conforms then

- $\neg$conforms
- Test Case Generator
- Abstract Test Case
- SUT
- Test Driver

then pass/fail
Model-Based Mutation Testing

- Model
- Mutation Tool
- Test Case Generator
- Abstract Test Case
- SUT
- Test Driver
Model-Based Mutation Testing

- Model
- Mutation Tool
- Model Mutant
- Test Case Generator
- Abstract Test Case
- SUT
- Test Driver

pass / fail

then pass/fail
then fail

¬ conforms
Model-Based Mutation Testing

if ¬conforms

Model → Mutation Tool → Model Mutant

Test Case Generator

Abstract Test Case

SUT

Test Driver

then pass/fail
Model-Based Mutation Testing

- Model
- Mutation Tool
- Model Mutant
- Test Case Generator
- Abstract Test Case
- SUT
- Test Driver

if \(\neg\)conforms

then fail

if conforms
Model-Based Mutation Testing

If \( \neg \text{conforms} \) then \( \neg \text{conforms} \)

If \( \neg \text{conforms} \) then pass

If \( \text{conforms} \) then pass/fail

If \( \neg \text{conforms} \) then fail

Then fail
Theorem

Given a transitive conformance relation \( \sqsubseteq \), then

\[
(Model \nsubseteq SUT) \land (Mutant \sqsubseteq SUT) \Rightarrow (Model \nsubseteq Mutant)
\]

- 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:
  \[ \text{Test case} \subseteq \text{SUT} \]
- If we generate a test case from a model, we have selected a partial behaviour such that
  \[ \text{Test case} \subseteq \text{Model} \]
- If SUT conforms to the model:
  \[ \text{Test case} \subseteq \text{Model} \subseteq \text{SUT} \]
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:
  \[ \text{Test case} \subseteq \text{SUT} \]

- If we generate a test case from a model, we have selected a partial behaviour such that
  \[ \text{Test case} \subseteq \text{Model} \]

- If SUT conforms to the model:
  \[ \text{Test case} \subseteq \text{Model} \subseteq \text{SUT} \]
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:
  \[
  \text{Test case} \sqsubseteq \text{SUT}
  \]

- If we generate a test case from a model, we have selected a partial behaviour such that
  \[
  \text{Test case} \sqsubseteq \text{Model}
  \]

- If SUT conforms to the model:
  \[
  \text{Test case} \sqsubseteq \text{Model} \sqsubseteq \text{SUT}
  \]
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:

\[ \text{Test case} \sqsubseteq \text{SUT} \]

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

\[ \text{Test case} \sqsubseteq \text{Model} \]

If SUT conforms to the model:

\[ \text{Test case} \sqsubseteq \text{Model} \sqsubseteq \text{SUT} \]
Fault-Detecting Test Case

- Generated from the model
- Kills the mutant

\[ \text{Test case} \sqsubseteq \text{Model} \]

- It is a counter-example to conformance, hence

\[ \text{Model} \not\sqsubseteq \text{Mutant} \]

\[ \text{iff} \]

\[ \exists \text{Test case} : (\text{Test case} \sqsubseteq \text{Model} \land \text{Test case} \not\sqsubseteq \text{Mutant}) \]

Fault-Detecting Test Case

- Generated from the model
- Kills the mutant

\[ \text{Test case} \sqsubseteq \text{Model} \land \text{Test case} \not\sqsubseteq \text{Mutant} \]

- It is a counter-example to conformance, hence

\[ \text{Model} \not\sqsubseteq \text{Mutant} \]

iff

\[ \exists \text{Test case} : (\text{Test case} \sqsubseteq \text{Model} \land \text{Test case} \not\sqsubseteq \text{Mutant}) \]

Fault-Detecting Test Case

- Generated from the model
- Kills the mutant

\[ \text{Test case} \sqsubseteq \text{Model} \land \text{Test case} \not\sqsubseteq \text{Mutant} \]

- It is a counter-example to conformance, hence

\[ \text{Model} \not\sqsubseteq \text{Mutant} \]

iff

\[ \exists \text{Test case} : (\text{Test case} \sqsubseteq \text{Model} \land \text{Test case} \not\sqsubseteq \text{Mutant}) \]

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 \( \text{Model}(s, s') \) and \( \text{Mutant}(s, s') \) describing state transformations \( (s \rightarrow s') \)
- Conformance:
  \[
  \text{Model} \sqsubseteq \text{Mutant} =_{df} \forall s, s': \text{Mutant}(s, s') \Rightarrow \text{Model}(s, s')
  \]
- Non-conformance:
  \[
  \text{Model} \nsubseteq \text{Mutant} = \exists s, s': \text{Mutant}(s, s') \land \neg \text{Model}(s, s')
  \]
- Read: a behaviour allowed by mutant but not by original model?
- This is a constraint satisfaction problem!
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:

  $$Model \nsubseteq Mutant = \exists s, s': Mutant(s, s') \land \neg Model(s, s')$$

- Read: a behaviour allowed by mutant but not by original model?

- This is a constraint satisfaction problem!

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:

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

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

Triangle semantics:

\[
\begin{align*}
\text{Mutant}(a, b, c, \text{res'}) \land \neg \text{Model}(a, b, c, \text{res'}) \equiv_{df} \\
(\ldots) \\
\neg(\ldots \neg(a \leq c - b \lor a \leq b - c \lor b \leq a - c) \land (a \geq b \land b = c \land \text{res'} = \text{equilateral}) \\
(\ldots) \\
\newline
\end{align*}
\]

- Simplifies to \(a > b \land b = c \land \text{res'} = \text{equilateral}\)
- Solver produces solution: \(a = 3, b = 2, c = 2, \text{res'} = \text{equilateral}\)
- Test case with expected result: \(a = 3, b = 2, c = 2, \text{res'} = \text{isosceles}\)
Transformational Systems: Tools

Implemented with different solvers:

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


Implemented with different solvers:

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

**References**

Transformational Systems: Tools

Implemented with different solvers:

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


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]

\[ SUT \text{ioco Model} \equiv df \]
\[ \forall \sigma \in \text{traces(Model)} : \]
\[ \text{out}(SUT \text{ after } \sigma) \subseteq \text{out(Model after } \sigma) \]

out ... outputs + quiescence
after ... reachable states after trace
Operational semantics
e.g. Labelled Transition Systems

Input-output conformance (ioco) [Tretmans96]

\[ SUT \text{ ioco } Model \equiv_{df} \]

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

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

- Operational semantics
e.g. Labelled Transition Systems

- Input-output conformance (ioco)
[Tretmans96]

$SUT \ ioco \ Model \ =_{df} \ \forall \sigma \in \text{traces}(Model) :$

$\text{out}(SUT \ \text{after} \ \sigma) \subseteq \text{out}(Model \ \text{after} \ \sigma)$

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

SUT $ioco$ Model $\checkmark$
Explicit Conformance Checking

- Model and Mutant $\rightarrow$ LTS
- Determinisation

Model:

- !flashOn
- !soundOn
- !soundOn
- !flashOn

Mutant:

- !flashOn
- !soundOff
- ?unlock

Build synchronous product modulo $\text{ioco}$

- If mutant has additional
  - !output: $\rightarrow$ fail sink state
  - ?input: $\rightarrow$ pass sink state

Model $\times_{\text{ioco}}$ Mutant:

- !flashOn
- !soundOn
- !soundOn
- !soundOff
- ?unlock

Extract test case covering fail state
Explicit Conformance Checking

- Model and Mutant $\rightarrow$ LTS
- Determinisation

**Model:**

- $\neg$flashOn
- $\neg$soundOn
- $\neg$soundOn
- $\neg$flashOn

**Mutant:**

- $\neg$flashOn
- $\neg$soundOff
- $\neg$unlock

- Build synchronous product modulo $\text{ioco}$

- If mutant has additional
  - $\neg$output: $\rightarrow$ fail sink state
  - $?input$: $\rightarrow$ pass sink state

**Model $\times_{\text{ioco}}$ Mutant:**

- $\neg$flashOn
- $\neg$soundOn
- $\neg$soundOn
- $\neg$soundOff
- $?unlock$

- Extract test case covering fail state
Explicit Conformance Checking

- Model and Mutant $\rightarrow$ LTS
- Determinisation

Model:

$\neg$flashOn $\rightarrow$ !soundOn

$\neg$flashOn $\rightarrow$ !soundOn $\rightarrow$

Mutant:

$\neg$flashOn $\rightarrow$ !soundOff

$\neg$flashOn $\rightarrow$ ?unlock

- Build synchronous product modulo $\mu\text{oco}$
- If mutant has additional
  - $!output$: $\rightarrow$ fail sink state
  - $?input$: $\rightarrow$ pass sink state

Model $\times_{\mu\text{oco}}$ Mutant:

!soundOn $\rightarrow$ pass

!soundOff $\rightarrow$ fail

!soundOn $\rightarrow$ pass

?unlock $\rightarrow$ pass

- 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


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


Applications of Explicit Conformance Checking

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

Scalability: abstractions for data-intensive models


Applications of Explicit Conformance Checking

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

Scalability: abstractions for data-intensive models


Applications of Explicit Conformance Checking

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

**Scalability**: abstractions for data-intensive models

---


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 :: ¬closed → closed := true;
Open :: closed → closed := false;
SoundOn :: armed ∧ ¬closed ∧ ¬sound → sound := true;
FlashOn :: armed ∧ ¬closed ∧ ¬flash → flash := true

... do Close □
   □ Open
   □ SoundOn; FlashOn
   □ 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 $tr$:

$$
\begin{align*}
  l :: g \rightarrow B &= df \ g \land B \land tr' = tr^\sim[l] \\
  l(\overline{x}) :: g \rightarrow B &= df \ \exists \overline{x} : g \land B \land tr' = tr^\sim[l(\overline{x})] \\
  x := e &= df \ x' = e \land y' = y \land \ldots \land z' = z \\
  g \rightarrow B &= df \ g \land B \\
  B(s, s') ; B(s, s') &= df \ \exists s_0 : B(s, s_0) \land B(s_0, s') \\
  B_1 \Box B_2 &= df \ B_1 \lor B_2
\end{align*}
$$

- then simplified (DNF + quantifier elimination)
Symbolic Conformance Checking

\[ \exists s, s', tr, tr' : \text{reachable}(s, tr) \land \text{Mutant}(s, s', tr, tr') \land \neg \text{Model}(s, s', tr, tr') \]

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

\[ \exists s_1, s'_1, s_2, s'_2, tr, !a : \text{reachable}(\text{Mutant}, tr, s_1) \land \text{reachable}(\text{Model}, tr, s_2) \land \text{Mutant}(s_1, s'_1, tr, tr \uparrow !a) \land \neg \text{Model}(s_2, s'_2, tr, tr \uparrow !a) \]
Symbolic Conformance Checking

\[ \exists s, s', tr, tr' : \text{reachable}(s, tr) \land \text{Mutant}(s, s', tr, tr') \land \neg \text{Model}(s, s', tr, tr') \]

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

\[ \exists s_1, s_1', s_2, s_2', tr, !a : \text{reachable} (\text{Mutant}, tr, s_1) \land \text{reachable} (\text{Model}, tr, s_2) \]
\[ \land \]
\[ \text{Mutant}(s_1, s_1', tr, tr \uparrow !a) \land \neg \text{Model}(s_2, s_2', tr, tr \uparrow !a) \]
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, 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:


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!


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.

- Explicit Checker: 65s
- 1st Symbolic Checker: 108s
- Quantifier Elimination: 41s
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.

- Explicit Checker: 65s
- Symbolic Checker: 108s
- Quantifier Elimination: 41s
- Variable/Value Selection: 27s
- Syntactic Mutation Localisation: 19s
- Incremental Solving: 2.8s
Optimisations

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

- Explicit Checker: 65s
- 1st Symbolic Checker: 108s
- Quantifier Elimination: 41s
- Variable/Value Selection: 27s
- Syntactic Mutation Localisation: 19s
- Incremental Solving: 2.8s
- Reachability Once: 2.6s
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

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
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

MoMuT::UML


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


Case Study 1: Car Alarm System

State machine model in UML

Metrics of Generated Action System

<table>
<thead>
<tr>
<th>CAS_UML</th>
</tr>
</thead>
<tbody>
<tr>
<td>actions [#]</td>
</tr>
<tr>
<td>state variables [#]</td>
</tr>
<tr>
<td>possible states [#]</td>
</tr>
<tr>
<td>reachable states [#]</td>
</tr>
<tr>
<td>required exploration depth</td>
</tr>
</tbody>
</table>
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):

<table>
<thead>
<tr>
<th></th>
<th>conforming (refining)</th>
<th>conforming (non-ref., but ioco)</th>
<th>not conforming (non-ref. &amp; not ioco)</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>mutants [#]</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>13</td>
<td>4</td>
<td>145</td>
</tr>
<tr>
<td>ref. check</td>
<td></td>
<td>4.03</td>
<td>1.63</td>
<td>56.41</td>
</tr>
<tr>
<td></td>
<td>Σ</td>
<td>0.31</td>
<td>0.41</td>
<td>0.39</td>
</tr>
<tr>
<td></td>
<td>φ</td>
<td>0.41</td>
<td>0.44</td>
<td>0.53</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>-</td>
<td>17.71</td>
<td>1.9 min</td>
</tr>
<tr>
<td>ioco check</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Σ</td>
<td>-</td>
<td>17.71</td>
<td>1.9 min</td>
</tr>
<tr>
<td></td>
<td>φ</td>
<td>-</td>
<td>4.43</td>
<td>0.79</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>-</td>
<td>4.48</td>
<td>2.01</td>
</tr>
<tr>
<td>tc constr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Σ</td>
<td>-</td>
<td>-</td>
<td>1.3 min</td>
</tr>
<tr>
<td></td>
<td>φ</td>
<td>-</td>
<td>-</td>
<td>0.55</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>-</td>
<td>-</td>
<td>1.48</td>
</tr>
<tr>
<td>total</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Σ</td>
<td>4.25</td>
<td>19.4</td>
<td>4.2 min</td>
</tr>
<tr>
<td></td>
<td>φ</td>
<td>0.33</td>
<td>4.85</td>
<td>1.74</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>0.43</td>
<td>4.89</td>
<td>2.77</td>
</tr>
</tbody>
</table>

**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

Metrics of Generated Action System

- actions [#]: 109
- state variables [#]: 74
- possible states [#]: $1.2 \times 10^{31}$
- reachable states [#]: > 850,700
- required exploration depth: > 25
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\(+5\)) :

<table>
<thead>
<tr>
<th></th>
<th>conforming (refining)</th>
<th>conforming (non-ref., but ioco)</th>
<th>not conforming (non-ref. &amp; not ioco)</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutants [#]</td>
<td></td>
<td></td>
<td></td>
<td>1185</td>
</tr>
<tr>
<td><strong>Σ</strong></td>
<td>6.1 h</td>
<td>189</td>
<td>7.7</td>
<td>13.3 h</td>
</tr>
<tr>
<td><strong>φ</strong></td>
<td>1.9</td>
<td>68</td>
<td>6.8 sec</td>
<td>7.1 h</td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>4.3</td>
<td>1.8</td>
<td>0.7 h</td>
<td>2.4 h</td>
</tr>
<tr>
<td>ref. check</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Σ</strong></td>
<td>-</td>
<td>0.7 h</td>
<td>1.7 h</td>
<td>2.4 h</td>
</tr>
<tr>
<td><strong>φ</strong></td>
<td>-</td>
<td>38 sec</td>
<td>7 sec</td>
<td></td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>-</td>
<td>2</td>
<td>27 sec</td>
<td></td>
</tr>
<tr>
<td>ioco check</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Σ</strong></td>
<td>-</td>
<td>-</td>
<td>22.9</td>
<td>22.9</td>
</tr>
<tr>
<td><strong>φ</strong></td>
<td>-</td>
<td>-</td>
<td>1.5 sec</td>
<td></td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>-</td>
<td>-</td>
<td>3.7 sec</td>
<td></td>
</tr>
<tr>
<td>tc constr.</td>
<td></td>
<td></td>
<td></td>
<td>16.2 h</td>
</tr>
<tr>
<td>total</td>
<td>6.1 h</td>
<td>68</td>
<td>0.9 h</td>
<td>9.2 h</td>
</tr>
<tr>
<td><strong>φ</strong></td>
<td>1.9</td>
<td>1.8</td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td><strong>max</strong></td>
<td>4.3</td>
<td>2.2</td>
<td>4.1</td>
<td></td>
</tr>
<tr>
<td>without logging</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Σ</strong></td>
<td></td>
<td></td>
<td>16.2 h</td>
<td></td>
</tr>
<tr>
<td><strong>φ</strong></td>
<td></td>
<td></td>
<td>0.8</td>
<td></td>
</tr>
<tr>
<td><strong>max</strong></td>
<td></td>
<td></td>
<td>4.3</td>
<td></td>
</tr>
</tbody>
</table>
Case Study 2: Run-times

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

<table>
<thead>
<tr>
<th></th>
<th>not ioco</th>
<th>ioco</th>
<th>total</th>
</tr>
</thead>
<tbody>
<tr>
<td>mutants [#]</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Σ</td>
<td>9.8 h</td>
<td>22.8 h</td>
<td>32.6 h</td>
</tr>
<tr>
<td>φ</td>
<td>0.8</td>
<td>2.9</td>
<td>1.7</td>
</tr>
<tr>
<td>max</td>
<td>3.9</td>
<td>5.2</td>
<td>5.2</td>
</tr>
<tr>
<td>time – tc constr.</td>
<td>19</td>
<td>-</td>
<td>19</td>
</tr>
<tr>
<td>Σ</td>
<td>10.1 h</td>
<td>22.8 h</td>
<td>32.9 h</td>
</tr>
<tr>
<td>φ</td>
<td>0.8</td>
<td>2.9</td>
<td>1.7</td>
</tr>
<tr>
<td>max</td>
<td>3.9</td>
<td>5.2</td>
<td>5.2</td>
</tr>
</tbody>
</table>

appr. 16h vs. 33h
Abstract Test Case of AVL489

Abstract test cases → concrete C# NUnit test cases.

ctr ... controllable event (input)
obsv ... observable event (output)
Test Execution on Particle Counter

We found several bugs in the SUT:

- Forbidden changes of operating state while busy
  - Pause $\rightarrow$ Standby
  - Normal Measurement $\rightarrow$ Integral Measurement
- Ignoring high-frequent input without error-messages
- Loss of error messages in client for remote control of the device
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 coffeefobutton =
      requires paid :
        coffee_sel := true ;
        paid := false ;
    end ;
    obs coffee =
      requires coffee_sel :
        skip
    end ;
do
  coin () [] coffeefobutton () [] coffee ()
od || system CoffeeMachine
```
Object-Oriented Action Systems:

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

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)
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)

- 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, coffeobutton;
Outputs  coffee, tea;
Internals  paid;

\{I\}  \text{not paid and not coffee and not tea}
\{R1\}  \text{assume coin'}
\text{guarantee paid'}
\{R2\}  \text{assume paid and teabutton' and not coffeobutton'}
\text{guarantee tea' and not paid'}
\{R3\}  \text{assume paid and coffeobutton' and not teabutton'}
\text{guarantee coffee' and not paid'}
\{R4\}  \text{assume teabutton' and coffeobutton'}
\text{guarantee skip}
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;

\{
\}
\not\ paid \ and \ not \ coffee \ and \ not \ tea

\{R1\}
\begin{align*}
\text{assume} & \ \text{coin}' \\
\text{guarantee} & \ \text{paid}'
\end{align*}

\{R2\}
\begin{align*}
\text{assume} & \ \text{paid} \ \text{and} \ \text{teabutton}' \ \text{and not} \ \text{coffeebutton}' \\
\text{guarantee} & \ \text{tea}' \ \text{and not} \ \text{paid}'
\end{align*}

\{R3\}
\begin{align*}
\text{assume} & \ \text{paid} \ \text{and} \ \text{coffeebutton}' \ \text{and not} \ \text{teabutton}' \\
\text{guarantee} & \ \text{coffee}' \ \text{and not} \ \text{paid}'
\end{align*}

\{R4\}
\begin{align*}
\text{assume} & \ \text{teabutton}' \ \text{and} \ \text{coffeebutton}' \\
\text{guarantee} & \ \text{skip}
\end{align*}
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 → 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 $\rightarrow$ 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].