Model-based Mutation Testing
The Science of Killing Bugs in a Black Box

Bernhard K. Aichernig

Institute for Software Technology
Graz University of Technology
Austria

Summer Term 2019
Background: Research

Research Area:

- Formal Development Methods
- Model-based Testing (MBT)

models → semantics → testing theories → test-case generators

Strong Collaboration:
Since 2008 with AIT
Since 2011 with AVL
Background: Projects

- **CREDO**: FP6, MBT of distributed systems
- **MOGENTES**: FP7, MBT of embedded systems, mutation testing, qualitative reasoning for testing hybrid systems
- **TRUFAL**: national, scalability of test-case generators via symbolic analysis
- **MBAT**: FP7, integration of methods and tools, MBT + consistency checking
- **CRYSTAL**: FP7, integration of tools, MBT + requirements engineering
- **TRUCONF**: national, MBT + non-functional requirements + systems of systems
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
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 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) >>> 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]
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

Source Code → **Mutation Process** → Mutant

Mutation Operator
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
    - \( a \geq b \Rightarrow \geq \)
  - creates 5 mutants

```java
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
    - $\Rightarrow$
  - 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: Reactive System

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

- Killed by
  - Lock();
  - Close();
  - After(20);
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 Mutation Testing**
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
Model-Based Mutation Testing

Diagram:

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

Logic:

- If conforms
  - Then ¬ conforms
- If ¬ conforms
  - Then conforms
- If conforms
  - Then fails
- If fails
  - Then pass
Non-Conformance & Test Cases

Theorem

Given a transitive conformance relation $\sqsubseteq$, then

$$(\text{Model} \not\sqsubseteq \text{SUT}) \land (\text{Mutant} \sqsubseteq \text{SUT}) \Rightarrow (\text{Model} \not\sqsubseteq \text{Mutant})$$

- 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:

\[
\text{Test case} \subseteq \text{Model} \subseteq \text{SUT}
\]
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 Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
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:

\[ \text{Mutant}(a, b, c, res') \land \neg \text{Model}(a, b, c, res') =_{df} \]

\[
\ldots
\neg(a \leq c - b \lor a \leq b - c \lor b \leq a - c) \land (a \geq b \land b = c \land res' = \text{equilateral})
\]

\[
\ldots \neg(a \leq c - b \lor a \leq b - c \lor b \leq a - c) \land (a = b \land b = c \land res' = \text{equilateral})
\]

- Simplifies to \( a > b \land b = c \land res' = \text{equilateral} \)
- 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)


Agenda

- Mutation Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
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
Operational semantics
e.g. Labelled Transition Systems

Input-output conformance (ioco) [Tretmans96]

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

out ... outputs + quiescence
after ... reachable states after trace
Explicit Conformance Checking

- Model and Mutant → LTS
- Determinisation

**Model:**

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

**Mutant:**

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

**Build synchronous product modulo ioco**

- If mutant has additional
  - !output: → fail sink state
  - ?input: → pass sink state

**Model × ioco Mutant:**

- !flashOn
- !soundOff
- ?unlock
- !soundOn pass
- !soundOn fail
- ?unlock 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


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 \( tr \):

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

- 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.
- loco for complete models:

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

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


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 Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
Agile Development

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

- Mutation Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
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 (cont.)

Applications:
- Car Alarm System (Ford)
- Railway Interlocking System (Thales)
- Automotive Measurement Device: Particle Counter (AVL)
SUT: AVL489 Particle Counter

- One of AVL’s automotive measurement devices
- Measures particle number concentrations in exhaust gas
- **Focus**: testing of the control logic
- AVL uses virtual test-beds with simulated devices for integration and regression testing.
- We tested a simulation of the particle counter:
  - Matlab Simulink model compiled to real-time executable
  - Same interface as real device!
**UML Test Model of AVL489**

**Pause_0**
- SetPurge [not Busy and not Manual]
- SetStandby [not Busy and not Manual]
- sent SPAU state /entry
- sent StatusBusy; set Busy /exit

**Standby_1**
- SetPurge [not Busy and not Manual]
- SetStandby [not Busy and not Manual]
- sent STBY_state /entry
- sent StatusBusy; set Busy /exit

**Active**
- Purging_Pause_12
  - send SPUL_state /entry
- Integral_9
  - send SINT_state /entry
- Measurement_2
  - send SMGA_state /entry
- ZeroGas_10
  - send SNGA_state /entry

**LeakageTest**
- sent SLEC_state /entry

**ResponseCheck**
- sent SEGA_state /entry

**Response_14**
- sent SPUL_state /entry

**Purging_Purging_Pause_12**
- sent SPUL_state /entry

**Integral_9**
- sent SINT_state /entry

**Measurement_2**
- sent SMGA_state /entry

**ZeroGas_10**
- sent SNGA_state /entry

**isReady**
- 30 [not (oclIsInState(Active::Response_14) or oclIsInState(Active::Purging_Purging_Pause_12) or oclIsInState(Active::Leakage_11) or oclIsInState(Active::ZeroGas_10) or oclIsInState(Active::Purging_Purging_Pause_12)) or not Busy - send StatusReady]

**isBusy**
- 30 [not (oclIsInState(Active::Response_14) or oclIsInState(Active::Purging_Purging_Pause_12) or oclIsInState(Active::Leakage_11) or oclIsInState(Active::ZeroGas_10) or oclIsInState(Active::Purging_Purging_Pause_12))]

**DilutionSelection**
- [not Manual and not Busy] / sent Dilution

**LeakageTest, ResponseCheck, SetPurge, SetZeroPoint, StopIntegralMeasurement, SetStandby, StartMeasurement, StartIntegralMeasurement, SetPause, DilutionSelection**

**SetStandby**
- [not Busy and not Manual]

**SetPurge**
- [not Busy and not Manual]

**SetPause**
- [not Busy and not Manual]

**SetZeroPoint**
- [not Busy and not Manual]

**StopIntegralMeasurement**
- [not Busy and not Manual]

**StartIntegralMeasurement**
- [not Busy and not Manual]

**StartMeasurement**
- [not Busy and not Manual]

**StopMeasurement**
- [not Busy and not Manual]

**StartMeasurement**
- [not Busy and not Manual]

**StartIntegralMeasurement**
- [not Busy and not Manual]

**StartIntegralMeasurement**
- [not Busy and not Manual]

**StopIntegralMeasurement**
- [not Busy and not Manual]

**StartMeasurement**
- [not Busy and not Manual]

**StartIntegralMeasurement**
- [not Busy and not Manual]
Abstract Test Case of AVL489

Abstract test cases $\rightarrow$ concrete C# NUnit test cases.

pass

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
**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


```plaintext
1  types
2    CoffeeMachine = autocons system
3  |
4    [ var
5      paid : Boolean = false ;
6      coffee__sel : Boolean = false
7    actions
8      ctr coin =
9        requires true :
10         paid := true
11       end;
12      ctr coffeebutton =
13        requires paid :
14          coffee__sel := true ;
15          paid := false ;
16       end ;
17      obs coffee =
18        requires coffee__sel :
19          skip
20       end ;
21  do
22    coin() [] coffeebutton() [] coffee()
23  od
24  ]] 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

![Timed Automata Diagram]

3 < x < 5
**coffee**!

2 < x < 3
**tea**!

coin?

coffeobutton?
\[x = 0\]
teabutton?
\[x = 0\]
MoMuT::TA (cont.)

- **tioco-conformance**: $M \text{ tioco } S$
  - $\text{out}(M) \subseteq \text{out}(S)$
  - 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


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;

{} 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
Agenda

- Mutation Testing
- Model-based Mutation Testing
- Transformational Systems
  - Semantics
  - Test Case Generation
- Reactive Systems
  - Semantics
  - Test Case Generation
- Model- and Test-Driven Development
- MoMuT Tools
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].