Killing Bugs in a Black Box with Model-based Mutation Testing

Bernhard K. Aichernig

Institute of Software Technology
Graz University of Technology, Austria

MT CPS Workshop
Vienna, 11 Apr 2016
Acknowledgements

Joint work with

J. Auer · H. Brandl · W. Herzner · E. Jöbstl · W. Krenn · R. Korosec ·
F. Lorber · D. Nickovic · A. Rosenmann · R. Schlick · B.V. Schmidt ·
M. Tappler · S. Tiran

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

Past:
- **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

Ongoing:
- **CRYSTAL**: FP7, integration of tools, MBT + requirements engineering
- **TRUCONF**: national, MBT + non-functional requirements + systems of systems
Agenda

- Model-based Mutation Testing
- Real-Time Systems
- Hybrid Systems
- Discrete Systems
Step 1: Create mutants
Step 2: Try to kill mutants

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

Quality of tests:
How many mutants survived? [Lipton71, Hamlet77, DeMillo et al.78]
Objective

Don’t write test cases,

generate them!
Objective

Don’t write test cases, generate them!
Timed Automata Model of a Car Alarm System

- Car alarm system model
- and a mutation representing a fault
- leading to non-conformance representing an observable failure
- resulting in a test case triggering this fault
- and propagating it to a visible failure
Timed Automata Model of a Car Alarm System

- Car alarm system model
- and a mutation representing a fault
- leading to non-conformance representing an observable failure
- resulting in a test case triggering this fault
- and propagating it to a visible failure
Timed Automata Model of a Car Alarm System

- Car alarm system model
- and a mutation representing a fault
- leading to non-conformance representing an observable failure
- resulting in a test case triggering this fault
- and propagating it to a visible failure
Timed Automata Model of a Car Alarm System

- Car alarm system model
- and a mutation representing a fault
- leading to non-conformance representing an observable failure
- resulting in a test case triggering this fault
- and propagating it to a visible failure
Timed Automata Model of a Car Alarm System

- Car alarm system model
- and a mutation representing a fault
- leading to non-conformance representing an observable failure
- resulting in a test case triggering this fault
- and propagating it to a visible failure

What is a failure?
Fault-Propagation in Models

Abstract 5-place buffer model:

Counter variable n is internal!
Fault-Propagation in Models

Let's inject a fault:

![State diagram](image)

How does this fault propagate?
A Good Test Case

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

Model-Based Testing

Test Case Generator

SUT

Test Driver

pass / fail
then pass
then pass/fail
then fail
then \( \neg \) conforms
Model-Based Testing

Model

Test Case Generator

SUT

Test Driver

- then pass
- then fail
- then ¬conforms

Test Driver: Conformance Checker

Abstract Test Case Generator
Model-Based Testing

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

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

Test Case Generator:

- Conformance Checker

B.K. Aichernig Killing Bugs in a Black Box with Model-based Mutation Testing
Model-Based Testing

if conforms

then pass
Model-Based Testing

if ¬conforms

SUT

Test Driver

Abstract Test Case

Test Case Generator

Model

Model Mutant

then pass/fail

then pass

B.K. Aichernig

Killing Bugs in a Black Box with Model-based Mutation Testing
Model-Based Mutation Testing

- Model
- Mutation Tool
- Test Case Generator: Conformance Checker
- Abstract Test Case
- SUT
- Test Driver

Test Case Generator:
- Conformance Checker

pass / fail
then pass / fail
then fail
then ¬ conforms
Model-Based Mutation Testing

Model → Mutation Tool → Model Mutant

Test Case Generator: Conformance Checker

Abstract Test Case

SUT → Test Driver
Model-Based Mutation Testing

if \( \neg \text{conforms} \)

\[ \text{then } \neg \text{conforms} \]

Abstract Test Case

Test Driver

then pass/fail
Model-Based Mutation Testing

- **Model**
- **Mutation Tool**
- **Model Mutant**
- **Test Case Generator**
  - **Conformance Checker**
- **Abstract Test Case**
- **SUT**
- **Test Driver**

if \(\neg\) conforms then fail

if conforms then pass

B.K. Aichernig Killing Bugs in a Black Box with Model-based Mutation Testing
Model-Based Mutation Testing

Model → Mutation Tool → Model Mutant

Model

if ¬conforms

Test Case Generator: Conformance Checker

Abstract Test Case

SUT

Test Driver

if conforms

then pass

then pass/fail

then fail

then ¬conforms

B.K. Aichernig

Killing Bugs in a Black Box with Model-based Mutation Testing

12 / 64
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 (UML state machines)
  - MoMuT::OOAS (OO Action Systems)
  - MoMuT::QAS (Qualitative Action Systems)
  - MoMuT::TA (Timed Automata)
  - MoMuT::TAS (Timed Action Systems)
  - MoMuT::REQs (Synchronous Requirement Interfaces)

www.momut.org
Agenda

- Model-based Mutation Testing
- Real-Time Systems
- Hybrid Systems
- Discrete Systems
Conformance Relation of Timed Systems

... defines in a testing theory what constitutes a failure.

**Definition (Timed input-output conformance – tioco [Krichen&Tripakis09])**

Given a timed automaton Model and a Mutant with inputs and outputs

\[
\text{Mutant tioco Model} \iff \forall \sigma \in L(\text{Model}) : \text{out(Mutant after } \sigma) \subseteq \text{out(Model after } \sigma)
\]

- \(S\) ... set of all states
- \(s_0\) ... initial state
- \(\sigma\) ... timed trace of labels
- \(\Sigma_O\) ... output labels

- \(A\) after \(\sigma\) = \(\{s \in S \mid s_0 \xrightarrow{\sigma} s\}\)
- \(\text{elapse}(s) = \{t > 0 \mid s \xrightarrow{t}\}\}
- \(\text{out}(s) = \{a \in \Sigma_O \mid s \xrightarrow{a}\} \cup \text{elapse}(s)\)
- \(\text{out}(S) = \bigcup_{s \in S} \text{out}(s)\)
tioco and Language Inclusion

**Theorem ([Krichen&Tripakis09])**

\[ L(Mutant) \subseteq L(Model) \Rightarrow \text{Mutant tioco Model} \]

**Theorem ([Krichen&Tripakis09])**

*If Model is input-enabled, then*

\[ \text{Mutant tioco Model} \Rightarrow L(Mutant) \subseteq L(Model) \]

For deterministic TA, reduce tioco check to language inclusion check (PSPACE-complete).

Demonic completion for deterministic TA

Σ_1 ∪ Σ_0
Construct a formula $\varphi_{A_I, A_S}^k$ that is satisfiable if $L(A_I) \not\subseteq L(A_S)$

providing a timed trace as witness

$$
\varphi_{A_I, A_S}^k \equiv \bigwedge_{i=1}^k (d^i \geq 0 \land 1 \leq \alpha^i \leq |\Sigma|) \land i \geq 1 \land i \leq k \land
\begin{align*}
\text{init}_{A_I}(X_I, C_I) \land \text{path}_{A_I}^{1,i-1}(A, D, X_I, C_I) \\
\text{init}_{A_S}(X_S, C_S) \land \text{path}_{A_S}^{1,i-1}(A, D, X_S, C_S) \\
\text{path}_{A_I}^{i,i}(A, D, X_I, C_I) \land \neg \text{path}_{A_S}^{i,i}(A, D, X_S, C_S)
\end{align*}
$$

(delays and actions) \land \text{(in i steps)} \land \text{(reach in mutant)} \land \text{(reach in model)} \land \text{(failure)}

Variable sets:
- $x^i \in X$ ... location at step $i$
- $\alpha^i \in A$ ... $i^{th}$ discrete action
- $d^i \in D$ ... $i^{th}$ time delay
- $\{c^i, c^*\} \subseteq C$ ... clock valuation after $i^{th}$ time and discrete step
Experimental Results I

- Bounded language inclusion check for deterministic Uppaal TA
- Implemented in Scala calling SMT solver Z3
- Car alarm system characteristics: deterministic,
  - 5 clock variables, 16 locations, 25 transitions.
- 8 mutation operators → 1,320 mutants
- Overall runtime: 30 minutes ($k = 12$)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Bounded Model Checking</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>12</td>
<td>1.4s</td>
<td>1.1s</td>
</tr>
</tbody>
</table>

Runtime details
Experimental Results I

- Bounded language inclusion check for deterministic Uppaal TA
- Implemented in Scala calling SMT solver Z3
- Car alarm system characteristics: deterministic,
  - 5 clock variables, 16 locations, 25 transitions.
- 8 mutation operators → 1,320 mutants
- Overall runtime: 30 minutes (k = 12)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Bounded Model Checking</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>12</td>
<td>1.4s</td>
<td>1.1s</td>
</tr>
</tbody>
</table>

Runtime details
Timed Action Systems

```
1 types{  
2   State = [ ... | Flash | FlashSound | Silent | SwitchOffAlarm | ... ]; }  
3 state{  
4   loc : State; }  
5 clocks[Real]{ c;d;e;f;g }  
6 init{  
7   loc := OpenAndUnlocked; }  
8 invariant{  
9   if loc == Flash then e <= 0;  
10  if loc == FlashSound then e <= 30;  
11  if loc == Silent then e <= 300;  
12  ... }  
13 actions{  
14   !soundOn#1() if loc == Flash && e == 0 then { loc := FlashSound; };  
15  !soundOff#1() if loc == FlashSound && e == 30 then { loc := Silent ; };  
16  ?unlock#6() resets g if loc == FlashSound && e < 30 then { loc := SwitchOffAlarm; };  
17  ... }  
```
Symbolic Execution of Timed Action Systems

\[ s_0 \]

\[ pc = \overline{pc} \]
\[ q_c = \{ e \mapsto \overline{d}, \ldots \} \]

path conditions ... blue
symbolic (clock) states ... red
Symbolic Execution of Timed Action Systems

\[\text{path conditions } \ldots \text{blue}\]
\[\text{symbolic (clock) states } \ldots \text{red}\]

\[s_0\]
\[pc = \overrightarrow{pc}\]
\[q_c = \{e \mapsto \overrightarrow{d}, \ldots\}\]

\[!\text{soundOn}\]

\[s_1\]
\[pc = \overrightarrow{pc} \land \text{Flash} = \text{Flash}\]
\[q = \{\text{loc} \mapsto \text{FlashSound}, \ldots\}\]
Symbolic Execution of Timed Action Systems

\[ p_c = \overrightarrow{p_c} \]
\[ q_c = \{ e \mapsto \overrightarrow{d}, \ldots \} \]

\[ p_c = \overrightarrow{p_c} \land Flash \mapsto Flash \]
\[ q = \{ loc \mapsto FlashSound, \ldots \} \]

Path conditions ... blue
Symbolic (clock) states ... red
Symbolic Execution of Timed Action Systems

\[ s_0 \]
\[ pc = \overrightarrow{pc} \]
\[ q_c = \{ e \mapsto \overrightarrow{d}, \ldots \} \]

!soundOn

\[ s_1 \]
\[ pc = \overrightarrow{pc} \]
\[ q = \{ \text{loc} \mapsto \text{FlashSound}, \ldots \} \]

 delay(d)

\[ s_2 \]
\[ pc = \overrightarrow{pc} \land \text{Flash} = \text{Flash} \rightarrow \overrightarrow{d} + d \leq 30 \land \ldots \]
\[ q_c = \{ e \mapsto \overrightarrow{d} + d, \ldots \} \]

path conditions ... blue
symbolic (clock) states ... red
Symbolic Execution of Timed Action Systems

\[ \begin{align*}
& s_0 \\
& pc = \overrightarrow{pc} \\
& q_c = \{ e \mapsto d, \ldots \}
\end{align*} \]

\[ \text{!soundOn} \]

\[ \begin{align*}
& s_1 \\
& pc = \overrightarrow{pc} \\
& q = \{ loc \mapsto \text{FlashSound}, \ldots \}
\end{align*} \]

\[ \text{delay}(d) \]

\[ \begin{align*}
& s_2 \\
& pc = \overrightarrow{pc} \land \overrightarrow{d} + d \leq 30 \\
& q_c = \{ e \mapsto d + d, \ldots \}
\end{align*} \]

path conditions \ldots blue
symbolic (clock) states \ldots red
Symbolic Execution of Timed Action Systems

\[
\begin{align*}
S_0 & \quad pc = \overline{pc} \\
S_1 & \quad pc = \overline{pc}, \quad q_c = \{e \mapsto d, \ldots\} \\
S_2 & \quad pc = \overline{pc} \land d + d \leq 30, \quad q_c = \{e \mapsto d + d, \ldots\} \\
S_3 & \quad pc = \overline{pc} \land d + d < 30, \quad q_c = \{g \mapsto 0\} \quad \text{(delay \(d'\))} \\
S_4 & \quad pc = \overline{pc} \land d + d = 30, \quad q = \{loc \mapsto Silent\} \quad \text{(delay \(d'\))}
\end{align*}
\]

path conditions ... blue  
symbolic (clock) states ... red
Symbolic Execution of Timed Action Systems

Provides all symbolic timed traces through model!
Conformance Checking via Symbolic Execution

- Bounded implicit product graph exploration
- Simultaneous symbolic execution of all model traces
- Non-conformance checks (stioco) of the form:

\[ \exists q_{\text{fail}} \in \text{ModelStates} \land \forall \text{all symbolic states after current trace}, \exists \lambda \in \text{Observations} : \]
\[ p_{c_q} \land \lambda q_{\text{fail}} \land (\lor s \in \text{MutantStates} : p_{c_s} \land \text{guards} \lambda [\text{state} s]) \land \lambda \text{observation possible (mutant)} \land \neg (\lor q \in \text{ModelStates} : p_{c_q} \land \text{guards} \lambda [\text{state} q]) \land \lambda \text{observation not possible (model)} \]

\[ p_{c_q} \ldots \text{path condition of symbolic state } q \]
Conformance Checking via Symbolic Execution

- Bounded implicit product graph exploration
- Simultaneous symbolic execution of all model traces
- Non-conformance checks (stioco) of the form:

\[ \exists q_{\text{fail}} \in \text{ModelStates} \]

all symbolic states after current trace

\[ pc_{q_{\text{fail}}} \]

state reachable (model)

\[ pc_q \ldots \text{path condition of symbolic state } q \]
Conformance Checking via Symbolic Execution

- Bounded implicit product graph exploration
- Simultaneous symbolic execution of all model traces
- Non-conformance checks (stioco) of the form:

\[ \exists q_{\text{fail}} \in \text{ModelStates}, \exists \lambda \in \text{Observations} : \]

\[ pc_{q_{\text{fail}}} \land \left( \bigvee_{s \in \text{MutantStates}} pc_s \land \text{guards}_\lambda[s] \right) \land \neg \left( \bigvee_{q \in \text{ModelStates}} pc_q \land \text{guards}_\lambda[q] \right) \]

\( pc_q \ldots \) path condition of symbolic state \( q \)
Experimental Results II

- Symbolic execution to check for deterministic Timed Action Systems
- Implemented in Scala calling SMT solver Z3
- Car alarm system characteristics: deterministic,
  - 5 clock variables, 16 locations, 25 transitions.
- 8 mutation operators → 986 mutants
- Overall runtime: 27.5 minutes ($k = 12$)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Bounded Model Checking</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>12</td>
<td>1.4s</td>
<td>1.1s</td>
</tr>
</tbody>
</table>
Experimental Results II

- Symbolic execution tioco check for deterministic Timed Action Systems
- Implemented in Scala calling SMT solver Z3
- Car alarm system characteristics: deterministic,
  - 5 clock variables, 16 locations, 25 transitions.
- 8 mutation operators $\rightarrow$ 986 mutants
- Overall runtime: 27.5 minutes ($k = 12$)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Bounded Model Checking</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>12</td>
<td>1.4s</td>
<td>1.1s</td>
</tr>
</tbody>
</table>

Runtime details
Experimental Results III

- Symbolic tioco checker also for non-deterministic models
- Car Alarm System: silent transition with non-deterministic delay
- Plus underspecification in switching on alarm
- 3 equivalent mutants timed out after 10min
Experimental Results III

- Symbolic tioco checker also for non-deterministic models
- Car Alarm System: silent transition with non-deterministic delay
- Plus underspecification in switching on alarm
- 3 equivalent mutants timed out after 10min

<table>
<thead>
<tr>
<th>Depth</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Mean 0.79s Median 0.06s Max 360.84s Min ~ 0s</td>
</tr>
</tbody>
</table>
Experimental Results III

- Symbolic ticoc checker also for non-deterministic models
- Car Alarm System: silent transition with non-deterministic delay
- Plus underspecification in switching on alarm
- 3 equivalent mutants timed out after 10min

<table>
<thead>
<tr>
<th>Depth</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Mean 0.79s, Median 0.06s, Max 360.84s, Min ~ 0s</td>
</tr>
</tbody>
</table>

... and the bounded model checking?
Bounded Determinisation of Timed Automata
Bounded Determinisation of Timed Automata

The diagram illustrates the transition between states, labeled with actions and conditions. For example, a transition labeled "coin {x}" indicates that a coin is inserted, and the next state depends on the value of x. The unfolding process is shown to split the automaton into multiple layers, each representing a deeper level of detail. The labels on the transitions describe the conditions under which the transition occurs, such as "0 < x < 3" or "1 < x < 4."
Bounded Determinisation of Timed Automata

\[ q_0 \xrightarrow{\epsilon} q_3, \quad x = 1 \quad \text{BREWING} \]
\[ q_1 \xrightarrow{\text{beep}} q_2, \quad 0 < x < 3 \quad \text{GRAINING} \]
\[ q_0 \xrightarrow{\text{refund}} q_4, \quad x < 4 \quad \text{HEATING} \]
\[ q_3 \xrightarrow{\epsilon} q_1, \quad 1 < x < 2 \quad \{x\} \]

\[ \text{unfolding} \]

\[ q_0 \xrightarrow{\text{beep}} q_1, \quad 0 < x < 3 \]
\[ q_1 \xrightarrow{\text{beep}} q_2, \quad x = 2 \]
\[ q_2 \xrightarrow{\epsilon} q_3, \quad 1 < x < 2 \quad \{x_2\} \]
\[ q_3 \xrightarrow{\text{coffee}} q_4, \quad x_2, 0 = 1 \quad \{x_3\} \]
\[ q_4 \xrightarrow{\epsilon} q_6, \quad x_1 < 4 \quad \{x_3\} \]

\[ \epsilon\text{-removal} \]

\[ q_0 \xrightarrow{\text{beep}} q_1, \quad 0 < x < 3 \]
\[ q_1 \xrightarrow{\text{beep}} q_2, \quad x = 2 \]
\[ q_2 \xrightarrow{\epsilon} q_3, \quad 1 < x < 2 \quad \{x_2\} \]
\[ q_3 \xrightarrow{\text{coffee}} q_4, \quad 2 < x < 3 \quad \{x_3\} \]
\[ q_4 \xrightarrow{\epsilon} q_5, \quad x_1 < 4 \quad \{x_3\} \]
Bounded Determinisation of Timed Automata

unfolding

ε-removal

determinisation

B.K. Aichernig

Killing Bugs in a Black Box with Model-based Mutation Testing
Experimental Results IV

- **Bounded determinization**
  - 13,545 locations (depth 12)
  - bounded model check fails

- **Partial models!**

### Model Comparison

<table>
<thead>
<tr>
<th>Model</th>
<th>D.</th>
<th>Bounded Model Checking</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>Partial 1</td>
<td>8</td>
<td>9.7s</td>
<td>8.0s</td>
</tr>
<tr>
<td>Partial 2</td>
<td>12</td>
<td>1.6s</td>
<td>1.63s</td>
</tr>
<tr>
<td>Complete</td>
<td>12</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Experimental Results IV

- **Bounded determinization**
  - 13,545 locations (depth 12)
  - bounded model check fails
- **Partial models!**

<table>
<thead>
<tr>
<th>Model</th>
<th>D.</th>
<th>Bounded Model Checking</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>Partial 1</td>
<td>8</td>
<td>9.7s</td>
<td>8.0s</td>
</tr>
<tr>
<td>Partial 2</td>
<td>12</td>
<td>1.6s</td>
<td>1.63s</td>
</tr>
<tr>
<td>Complete</td>
<td>12</td>
<td>x</td>
<td>x</td>
</tr>
</tbody>
</table>
Experimental Results IV

- **Bounded determinization**
- 13,545 locations (depth 12)
- bounded model check fails
- **Partial models!**

<table>
<thead>
<tr>
<th>Model</th>
<th>D.</th>
<th>Bounded Model Checking</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partial 1</td>
<td>8</td>
<td>Mean: 9.7s Median: 8.0s Max: 85.1s Min: 0.3s</td>
<td>Mean: 0.28s Median: 0.04s Max: 16.78s Min: ~ 0s</td>
</tr>
<tr>
<td>Partial 2</td>
<td>12</td>
<td>Mean: 1.6s Median: 1.63s Max: 37.3s Min: 0.08s</td>
<td>Mean: 0.08s Median: 0.03s Max: 2.28s Min: ~ 0s</td>
</tr>
<tr>
<td>Complete</td>
<td>12</td>
<td>x x x x x</td>
<td>Mean: 0.79s Median: 0.06s Max: 360.84s Min: ~ 0s</td>
</tr>
</tbody>
</table>
Experimental Results V

- Adding **data variable and parameters** to
  - deterministic Car Alarm System with one clock
  - 3-digit **PIN** code for unlocking
- No negative effects, even with higher digit PIN codes
- Symbolic execution faster with 1 clock (0.24s) than with 5 clocks (1.7s)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Bounded Model Checking</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>8</td>
<td>1.46s</td>
<td>0.28s</td>
</tr>
<tr>
<td>12</td>
<td>4.12s</td>
<td>0.35s</td>
</tr>
</tbody>
</table>
Experimental Results V

- Adding **data variable and parameters** to
  - deterministic Car Alarm System with one clock
  - 3-digit PIN code for unlocking
- No negative effects, even with higher digit PIN codes
- Symbolic execution faster with 1 clock (0.24s) than with 5 clocks (1.7s)

<table>
<thead>
<tr>
<th>Depth</th>
<th>Bounded Model Checking</th>
<th>Symbolic Execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>Median</td>
</tr>
<tr>
<td>8</td>
<td>1.46s</td>
<td>0.28s</td>
</tr>
<tr>
<td>12</td>
<td>4.12s</td>
<td>0.35s</td>
</tr>
</tbody>
</table>
Real-Time Systems Summary

Symbolic execution (SE) seems to perform better, but no clear winner!

- **Number of clocks:**
  - BMC: small impact (was faster in deterministic case)
  - SE: high impact

- **Non-determinism:** is an obstacle for conformance checking
  - BMC: state-space explosion $\rightarrow$ partial models
  - SE: lowered performance (40s vs. 6min) $\rightarrow$ 3 mutants timed out

- **Statistical outliers:** due to equivalent mutants
  - BMC: runtime almost equal
  - SE: extreme differences due to optimisations
Agenda

- Model-based Mutation Testing
- Real-Time Systems
- Hybrid Systems
- Discrete Systems
A Hybrid System: Two Tank System

Requirements:
- P1 starts pumping, if T2 below Reserve and T1 is full
- until T1 is empty or T2 is full
- P2 is controlled by button WaterRequest
- runs if there is water in T2.
- Note: T1 may overflow

P1, P2 ... water pumps
G1, G2 ... water-level sensors
Related Work

▶ Hybrid Systems
  ▶ Hybrid Automata (Alur, Courcoubetis, Henzinger, Ho 93)
  ▶ Action Systems [Back, Kurki-Suonio 83]
  ▶ Hybrid Action Systems [Rönkkö, Ravn, Sere 03]
  ▶ Qualitative Reasoning [Kuipers 94]

▶ Testing
  ▶ Mutation Testing [Hamlet 77, De Millo et al. 78]
  ▶ Input-Output Conformance [Brinksma, Tretmans 92]
Abstraction 1: Action Systems

Modeling the Controller

Controller:

| var P1_running, P2_running : Bool, out*, inout* : Real |
| P1_running := false; P2_running := false; out := 0; inout := 0; |
| do |
| g1 → P1_running := true; inout := (0, Max] |
| g2 → P1_running := false; inout := 0 |
| g3 → P2_running := true; out := (0, Max] |
| g4 → P2_running := false; out := 0 |
| od |
| : WaterRequest, x1, x2 |

Guards:

- \( g_1 = \text{df} \) \( x_2 \leq \text{Reserve} \land x_1 = \text{Full} \land \neg P1\_running \)
- \( g_2 = \text{df} \) \( P1\_running \land (x_1 \leq \text{Empty} \lor x_2 = \text{Full}) \)
- \( g_3 = \text{df} \) \( \text{WaterRequest} \land \neg P2\_running \land x_2 > \text{Reserve} \)
- \( g_4 = \text{df} \) \( P2\_running \land (\neg \text{WaterRequest} \lor x_2 = \text{Empty}) \)
Abstraction 1: Action Systems

Modeling the Controller

Controller:

\[
\begin{align*}
\text{Controller:} & \quad |\begin{array}{l}
\textbf{var} \quad P1\_running, P2\_running : \text{Bool}, \\
\text{out}\^*, \text{inout}\^* : \text{Real} \\
\text{inout} := 0; \text{inout} := 0; \\
\text{do} \\
\quad g_1 \rightarrow P1\_running := \text{true}; \text{inout} := (0, \text{Max}] \\
\quad g_2 \rightarrow P1\_running := \text{false}; \text{inout} := 0 \\
\quad g_3 \rightarrow P2\_running := \text{true}; \text{out} := (0, \text{Max}] \\
\quad g_4 \rightarrow P2\_running := \text{false}; \text{out} := 0 \\
\text{od} \\
\textbf{out} & \quad : \quad \text{WaterRequest, } x_1, x_2
\end{array}\end{align*}
\]

Guards:

\[
\begin{align*}
\begin{array}{l}
\quad g_1 &= \text{df} \quad x_2 \leq \text{Reserve} \wedge \\
& \quad x_1 = \text{Full} \wedge \\
& \quad \neg P1\_running \\
\quad g_2 &= \text{df} \quad P1\_running \wedge \\
& \quad (x_1 \leq \text{Empty} \vee x_2 = \text{Full}) \\
\quad g_3 &= \text{df} \quad \text{WaterRequest} \wedge \\
& \quad \neg P2\_running \wedge \\
& \quad x_2 > \text{Reserve} \\
\quad g_4 &= \text{df} \quad P2\_running \wedge \\
& \quad (\neg \text{WaterRequest} \vee \\
& \quad x_2 = \text{Empty})
\end{array}
\end{align*}
\]
Why Action Systems?

▶ Well-suited for embedded systems modeling
▶ Action view maps naturally to LTS testing theories
▶ Solid foundation:
  ▶ precise semantics
  ▶ refinement
▶ Compositional modeling
▶ Many extensions available:
  ▶ object-orientation
  ▶ hybrid systems
Hybrid Action Systems

Environment:

\[\begin{array}{l}
\text{var} \quad x_1^*, x_2^* : \text{Real} \\
\text{\bullet} \\
x_1 := 0; x_2 := 0 \\
\text{alt} \\
g_1 \rightarrow \ldots \\
\square \\
\ldots \\
\text{with} \\
\neg (g_1 \lor \ldots) :\rightarrow \dot{x}_1 = (\text{in} - \text{inout})/A_1 \land \dot{x}_2 = (\text{inout} - \text{out})/A_2 \\
\end{array}\]

\[\begin{array}{l}
\text{||} \\
\text{out, inout} \\
\end{array}\]

Abstraction 2: Qualitative Flows

v-abs.f.t  f.t

max
high
med
zero
g.s

t-abs.f.t

t-zero
med
high
max
g.s

B.K. Aichernig
Killing Bugs in a Black Box with Model-based Mutation Testing
Example Qualitative Flow of Water Tanks
Qualitative Reasoning (QR)

- QR originates from Artificial Intelligence
- Common sense reasoning about physical systems with possibly incomplete knowledge.
- Ordinary Differential Equations (ODE)
  → Qualitative Differential Equations (QDE):
  \[ \dot{x}_1 = \frac{(in - inout)}{A_1} \rightarrow \frac{d}{dt}(x_1, \text{diff}_1) \land \text{add}(\text{diff}_1, inout, in) \]
- Arithmetic is reduced to sign algebra:
  \[ 5 - 1 = 4 \rightarrow [+] + [-] = [+] \mid [-] \]
  \[ -3 \times 2 = -6 \rightarrow [-] \times [+] = [-] \]
Qualitative Action Systems

```
[[
  var  \(x_1^*, x_2^* : \text{Real}\)
  
  \(\bullet\)
  \(x_1 := 0; x_2 := 0\)

  alt
  
  \(g_1 \rightarrow \ldots\)
  
  \(\square\)
  
  \(\ldots\)

  with
  
  \(\neg(g_1 \lor \ldots) \rightarrow\)
  
  \(d/dt(x_1, \text{diff}_1) \land d/dt(x_2, \text{diff}_2) \land\)
  
  \(\text{add}\text{diff}_2, \text{out}, \text{inout}) \land \text{adddiff}_1, \text{inout}, \text{in})\)

]]  :  \text{inout, out}
```
Implementations:
- QSIM (Lisp)
- Garp3 (SWI-Prolog)
- ASIM (GNU-Prolog)
Model-based Mutation Testing

Action System Model

\[ IOLTS^S \]

\[ IOLTS^M \]

discriminating test case

\[ \text{ioco?} \]

\[ \text{ioco} \ldots \text{input-output conformance} \]
Model-based Mutation Testing

Action System Model

IOLTS^S

discriminating test case

for every mutant

IOLTS^M

ioco... input-output conformance
Model-based Mutation Testing

Action System Model

\[ IOLTS^S \]

for every mutant

\[ IOLTS^M \]

discriminating test case

ioco ... input-output conformance
Model-based Mutation Testing

Action System Model

\[ IOLTS^S \]

discriminating test case

\[ IOLTS^M \]

for every mutant

\[ ioco? \]

\[ ioco \ldots \text{input-output conformance} \]
Conformance Checking

- Event-view: labeled actions
- Input and Output Labels

**Def. IOCO [Tretmans 96]**

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

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

- ioco supports: partial, non-deterministic models
  - ioco-checker Ulysses
    - implemented in GNU Prolog
    - explores discrete actions + qualitative flows
    - builds synchronous product modulo ioco
    - highly non-deterministic \(\rightarrow\) on-the-fly determinization
Conformance Checking

▶ Event-view: labeled actions
▶ Input and Output Labels

Def. IOCO [Tretmans 96]

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

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

▶ ioco supports: partial, non-deterministic models
▶ ioco-checker Ulysses
  ▶ implemented in GNU Prolog
  ▶ explores discrete actions + qualitative flows
  ▶ builds synchronous product modulo ioco
  ▶ highly non-deterministic \(\rightarrow\) on-the-fly determinization
Generating a Testcase: Original Model

\[
\text{System = }
\]
\[
\begin{align*}
\text{\texttt{\var{x_1}: T1, \var{x_2}: T2, \var{out}, \var{inout}: FR,}} \\
\text{\var{diff_1, diff_2}: NZP,} \\
\text{\var{p1\_running, p2\_running, wr}: Bool} \\
\bullet \quad \var{x_1} := (0, 0); \var{x_2} := (0, 0); \\
\text{out := (0, 0); inout := (0, 0); wr := false} \\
\var{p1\_running} := \text{false}; \var{p2\_running} := \text{false} \\
\text{alt} \\
\text{obs pump1\_on: } g_1 \rightarrow \var{p1\_running} := \text{true;} \\
\text{inout := (0..Max, 0)} \\
\text{□ obs pump1\_off: } g_2 \rightarrow \var{p1\_running} := \text{false;} \\
\text{inout := (0, 0)} \\
\text{□ obs pump2\_on: } g_3 \rightarrow \var{p2\_running} := \text{true;} \\
\text{out := (0..Max, 0)} \\
\text{□ obs pump2\_off: } g_4 \rightarrow \var{p2\_running} := \text{false;} \\
\text{out := (0, 0)} \\
\text{□ ctr water\_req(X): } g_5 \rightarrow \var{wr} := X \\
\text{with} \\
\neg (g_1 \lor g_2 \lor g_3 \lor g_4 \lor g_5) : \rightarrow \\
\text{add(\var{diff_2, out, inout})} \land \text{add(\var{diff_1, inout, in})} \land \\
\frac{d}{dt}(\var{x_1, diff_1}) \land \frac{d}{dt}(\var{x_2, diff_2}) \\
\end{align*}
\]
\[
\text{\texttt{\var{}}} : \text{in}
\]
Generating a Testcase II: Mutated Model

\[
\text{System = } \\
\begin{align*}
&\text{var } x_1 : T1, x_2 : T2, \text{ out, inout : FR,} \\
&\text{diff}_1, \text{ diff}_2 : \text{NZP,} \\
&\text{p1running, p2running, wr : Bool} \\
\end{align*} \\
\begin{align*}
&x_1 := (0, 0); \ x_2 := (0, 0) ; \\
&\text{out} := (0, 0); \ \text{inout} := (0, 0); \ \text{wr} := \text{false} \\
&\text{p1running} := \text{false}; \ \text{p2running} := \text{false} \\
\end{align*} \\
\begin{align*}
\text{alt} \\
&\text{obs pump1 on: } g_1 \rightarrow \text{p1running} := \text{true}; \\
&\text{inout} := (0..\text{Max}, 0) \\
\square \ &\text{obs pump1 off: } g_2 \rightarrow \text{p1running} := \text{true}; \\
&\text{inout} := (0, 0) \\
\square \ &\text{obs pump2 on: } g_3 \rightarrow \text{p2running} := \text{true}; \\
&\text{out} := (0..\text{Max}, 0) \\
\square \ &\text{obs pump2 off: } g_4 \rightarrow \text{p2running} := \text{false}; \\
&\text{out} := (0, 0) \\
\square \ &\text{ctr water req(X): } g_5 \rightarrow \text{wr} := X \\
\text{ with } \\
&\neg(g_1 \lor g_2 \lor g_3 \lor g_4 \lor g_5) := \rightarrow \\
&\text{add}(\text{diff}_2, \text{out, inout}) \land \text{add}(\text{diff}_1, \text{inout, in}) \land \\
&d/dt(x_1, \text{diff}_1) \land d/dt(x_2, \text{diff}_2) \\
\end{align*} \\
\]
Generating a Testcase III: Product Graph

Part of the result of the conformance check between the original and the mutated specification.
## Results

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ASO</td>
<td>10</td>
<td>13.9</td>
<td>64</td>
<td>117</td>
<td>7</td>
<td>3</td>
</tr>
<tr>
<td>ENO</td>
<td>6</td>
<td>7.6</td>
<td>68</td>
<td>120</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>ERO</td>
<td>20</td>
<td>12.9</td>
<td>62</td>
<td>110</td>
<td>20</td>
<td>0</td>
</tr>
<tr>
<td>LRO</td>
<td>13</td>
<td>12.8</td>
<td>93</td>
<td>168</td>
<td>9</td>
<td>4</td>
</tr>
<tr>
<td>MCO</td>
<td>16</td>
<td>12.8</td>
<td>70</td>
<td>126</td>
<td>10</td>
<td>6</td>
</tr>
<tr>
<td>RRO</td>
<td>12</td>
<td>12.0</td>
<td>40</td>
<td>73</td>
<td>10</td>
<td>2</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>77</strong></td>
<td><strong>12.0</strong></td>
<td><strong>66</strong></td>
<td><strong>119</strong></td>
<td><strong>61</strong></td>
<td><strong>16</strong></td>
</tr>
</tbody>
</table>

ASO ... Association Shift Operator  
ENO ... Expression Negation Operator  
ERO ... Event Replacement Operator  
LRO ... Logical Operator Replacement  
MCO ... Missing Condition Operator  
RRO ... Relational Replacement Operator
Generating a Testcase IV: Linear TC

Selecting one path for each unsafe state leading to failure.

- obs qual([x1:full/inc,x2:zero/std])
- obs out_pump1_on
- obs qual([x1:empty..full/inc,x2:full/inc])
- obs qual([x1: ...,x2: ...])
- obs out_pump2_on
- obs qual([x1:empty..full/inc,x2:empty..reserve/dec])
Generating a Testcase V: Adaptive TC

A test graph including all paths to a given unsafe state leading to failure.

Qualitative events are internal (not visible).
Hybrid Systems Summary

- AI meets FM: qualitative reasoning
- Requirements $\rightarrow$ incomplete qualitative models
- Model exploration: controller (discrete) + environment (qualitative)
- TCG based on mutation testing and ioco conformance checking
- Different strategies for selecting test case
Agenda

- Model-based Mutation Testing
- Real-Time Systems
- Hybrid Systems
- Discrete Systems
Discrete Systems: MoMuT::UML

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!
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!
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!
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
Abstract Test Case of AVL489

Abstract test cases → concrete C# NUnit test cases.

ctr ... controllable event (input)
obsw ... observable event (output)
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
Refinement + ioco Conformance Checking

Refinement:
- state-based
- predicative semantics

_Input-Output Conformance:_
- event-based
- io labelled transition systems

**Def. Refinement [Hoare & He 98]**
\[
\forall s, s' : \text{Mutant}(s, s') \Rightarrow \text{Model}(s, s')
\]

\(s\) ... state before
\(s'\) ... state after execution

**Def. IOCO [Tretmans 96]**
\[
\forall \sigma \in \text{traces}(\text{Model}) : \\
\text{out}(\text{Mutant after } \sigma) \subseteq \text{out}(\text{Model after } \sigma)
\]

\(\text{out}\) ... outputs labels + quiescence
after ... reachable states after trace

New combined conformance checking:
- Refinement checker searches for faulty state (fast)
- ioco checker looks if faulty state propagates to different observations
Refinement + ioco Conformance Checking

Refinement:
- state-based
- predicative semantics

Input-Output Conformance:
- event-based
- io labelled transition systems

Def. Refinement [Hoare & He 98]
\[ \forall s, s': \text{Mutant}(s, s') \Rightarrow \text{Model}(s, s') \]
s ... state before
s' ... state after execution

Def. IOCO [Tretmans 96]
\[ \forall \sigma \in \text{traces(Model)} : \]
out(\text{Mutant after } \sigma) \subseteq out(\text{Model after } \sigma)
out ... outputs labels + quiescence
after ... reachable states after trace

New combined conformance checking:
- Refinement checker searches for faulty state (fast)
- ioco checker looks if faulty state propagates to different observations
Refinement + ioco Conformance Checking

Refinement:
- state-based
- predicative semantics

Input-Output Conformance:
- event-based
- io labelled transition systems

Def. Refinement [Hoare & He 98]
\[ \forall s, s': \text{Mutant}(s, s') \Rightarrow \text{Model}(s, s') \]

Def. IOCO [Tretmans 96]
\[ \forall \sigma \in \text{traces(\text{Model})} : \]
\[ \text{out(\text{Mutant after } \sigma) } \subseteq \text{out(\text{Model after } \sigma)} \]

New combined conformance checking:
- Refinement checker searches for faulty state (fast)
- loco checker looks if faulty state propagates to different observations
Symbolic Refinement Checking

Is non-refinement reachable?

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

- \( s \ldots \text{state before} \)
- \( s' \ldots \text{states after execution} \)
- \( tr \ldots \text{trace of labels before} \)
- \( tr' \ldots \text{trace of labels after execution} \)
Institute of Software Technology

TCG Particle Counter

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

(b) Breakup into unique and duplicate test cases.

(c) Lengths of the unique test cases.

not conforming (non-ref. & not ioco)
conforming (refining)
conforming (non-ref., but ioco)

unique TCs
duplicate TCs

928
189
68
111
817

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

(2) Breakup into unique and duplicate test cases.

(3) Lengths of the unique test cases.
Fault Propagation

Figure: Number of steps from fault to failure (ioco depths)
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>189</td>
<td>68</td>
<td>928</td>
<td>1185</td>
</tr>
<tr>
<td>ref. check</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>6.1 h</td>
<td>7.7</td>
<td>7.1 h</td>
<td>13.3 h</td>
</tr>
<tr>
<td>$\phi$</td>
<td>1.9</td>
<td>6.8 sec</td>
<td>27 sec</td>
<td>40 sec</td>
</tr>
<tr>
<td>max</td>
<td>4.3</td>
<td>1.8</td>
<td>3.9</td>
<td>4.3</td>
</tr>
<tr>
<td>ioco check</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>-</td>
<td>0.7 h</td>
<td>1.7 h</td>
<td>2.4 h</td>
</tr>
<tr>
<td>$\phi$</td>
<td>-</td>
<td>38 sec</td>
<td>7 sec</td>
<td>7.4 sec</td>
</tr>
<tr>
<td>max</td>
<td>-</td>
<td>2</td>
<td>27 sec</td>
<td>2</td>
</tr>
<tr>
<td>tc constr.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\Sigma$</td>
<td>-</td>
<td>-</td>
<td>22.9</td>
<td>22.9</td>
</tr>
<tr>
<td>$\phi$</td>
<td>-</td>
<td>-</td>
<td>1.5 sec</td>
<td>1.2 sec</td>
</tr>
<tr>
<td>max</td>
<td>-</td>
<td>-</td>
<td>3.7 sec</td>
<td>3.7 sec</td>
</tr>
<tr>
<td>total</td>
<td>$\Sigma$</td>
<td>6.1 h</td>
<td>0.9 h</td>
<td>9.2 h</td>
</tr>
<tr>
<td></td>
<td>$\phi$</td>
<td>1.9</td>
<td>0.8</td>
<td>0.6</td>
</tr>
<tr>
<td></td>
<td>max</td>
<td>4.3</td>
<td>2.2</td>
<td>4.1</td>
</tr>
<tr>
<td>total without logging</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>$\Sigma$</td>
<td>6.1 h</td>
<td>0.9 h</td>
<td>9.2 h</td>
</tr>
</tbody>
</table>
Run-times

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

<table>
<thead>
<tr>
<th>Mutants [#]</th>
<th>Not ioco</th>
<th>Ioco</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mutants [#]</td>
<td>719</td>
<td>466</td>
<td>1185</td>
</tr>
<tr>
<td>Time – ioco check</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>9.8 h</td>
<td>22.8 h</td>
<td>32.6 h</td>
</tr>
<tr>
<td>Min</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>19</td>
<td>-</td>
<td>19</td>
</tr>
<tr>
<td>Min</td>
<td>1.6 sec</td>
<td>-</td>
<td>1 sec</td>
</tr>
<tr>
<td>Max</td>
<td>5.8 sec</td>
<td>-</td>
<td>5.8 sec</td>
</tr>
<tr>
<td>Total without logging</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Sum</td>
<td>10.1 h</td>
<td>22.8 h</td>
<td>32.9 h</td>
</tr>
<tr>
<td>Min</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
Discrete Systems Summary

- **Fault propagation** important for test-case design
- **Faster** test-case generator
  - find fault fast (refinement check)
  - analyze if fault propagates to failure (ioco check)
- **Optimized refinement check**
  - incremental SMT solving, state caching
  - exploiting the location of mutation
  - checking if existing test cases cover next fault
- **Applied at AVL**: many bugs found [TAP 2014]
Fault propagation important for test-case design

Faster test-case generator
  - find fault fast (refinement check)
  - analyze if fault propagates to failure (ioco check)

Optimized refinement check
  - incremental SMT solving, state caching
  - exploiting the location of mutation
  - checking if existing test cases cover next fault

Applied at AVL: many bugs found [TAP 2014]
Discrete Systems Summary

- **Fault propagation** important for test-case design
- **Faster** test-case generator
  - find fault fast (refinement check)
  - analyze if fault propagates to failure (ioco check)
- **Optimized** refinement check
  - incremental SMT solving, state caching
  - exploiting the location of mutation
  - checking if existing test cases cover next fault
- **Applied** at AVL: many bugs found [TAP 2014]
Synchronous Systems – MoMuT::REQs

Contract-based Requirement Interfaces:

- Synchronous assume-guarantee pairs
- Combined via conjunction
- Efficient SMT solving

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, Klaus Hörmaier, Florian Lorber, Dejan Nickovic, Rupert Schlick, Didier Simoneau, Stefan Tiran. Integration of Requirements Engineering and Test-Case Generation via OSLC, QSIC 2014
Synchronous Systems – MoMuT::REQs

Contract-based Requirement Interfaces:

- Synchronous assume-guarantee pairs
- Combined via conjunction
- Efficient SMT solving

Application: Airbag Chip (Infineon)

Inputs coin, teabutton, coffeebutton;

Outputs coffee, tea;

Internals paid;

\{I\} \texttt{not paid and not coffee and not tea}
\{R1\} \texttt{assume coin' guarantee paid'}
\{R2\} \texttt{assume paid and teabutton' and not coffeebutton' guarantee tea' and not paid'}
\{R3\} \texttt{assume paid and coffeebutton' and not teabutton' guarantee coffee' and not paid'}
\{R4\} \texttt{assume teabutton' and coffeebutton' guarantee skip}
Summary

▶ **Model-based Mutation Testing**
  ▶ Automatically test against anticipated faults
  ▶ TCG via conformance checks

▶ Real-Time Systems: Timed Automata
▶ Hybrid Systems: Action Systems + Qualitative Reasoning
▶ Discrete Systems: UML
▶ Synchronous Systems: Assume-Guarantee Contracts
▶ Ongoing projects:
  ▶ DSL for easier modelling, performance testing (AVL)
  ▶ Event-B refinement checker including sets, maps (Thales)
  ▶ Dependable Internet of Things: test-based model learning
Summary

- **Model-based Mutation Testing**
  - Automatically test against anticipated faults
  - TCG via conformance checks
- **Real-Time Systems**: Timed Automata
- **Hybrid Systems**: Action Systems + Qualitative Reasoning
- **Discrete Systems**: UML
- **Synchronous Systems**: Assume-Guarantee Contracts

**Ongoing projects:**
- DSL for easier modelling, performance testing (AVL)
- Event-B refinement checker including sets, maps (Thales)
- Dependable Internet of Things: test-based model learning
Summary

- **Model-based Mutation Testing**
  - Automatically test against anticipated faults
  - TCG via conformance checks
- **Real-Time Systems**: Timed Automata
- **Hybrid Systems**: Action Systems + Qualitative Reasoning
- **Discrete Systems**: UML
- **Synchronous Systems**: Assume-Guarantee Contracts
- **Ongoing projects**:
  - DSL for easier modelling, performance testing (AVL)
  - Event-B refinement checker including sets, maps (Thales)
  - **Dependable Internet of Things**: test-based model learning
References

Real-Time Systems


Hybrid Systems


Discrete Systems
