HALMSTAD UNIVERSITY

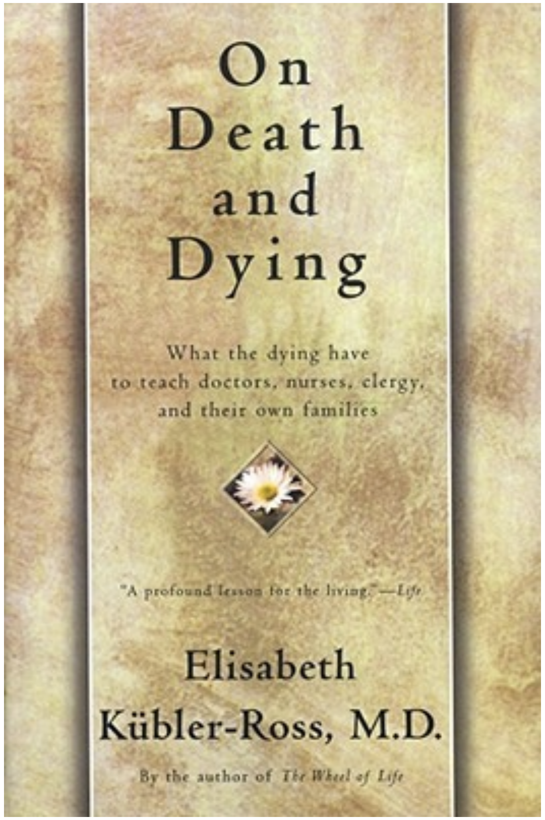
AUTO-CAAS: Model-Based Fault Prediction and Diagnosis of Automotive Software

Wojciech Mostowski and Mohammad Mousavi
Model-Based Testing Group,
Centre for Research on Embedded Systems (CERES)

Scandinavian Safety Conference 2016

Elevator pitch

- Bug fixing is like dying:
  Denial ➔ Anger ➔ Acceptance
- Demonstrating probability and severity to facilitate the process
- Using machine learning to capture all failing scenarios
- Context: AUTOSAR software



On
Death
and
Dying

What the dying have
to teach doctors, nurses, clergy,
and their own families

"A profound lesson for the living." —*Life*

Elisabeth
Kübler-Ross, M.D.

By the author of *The Wheel of Life*

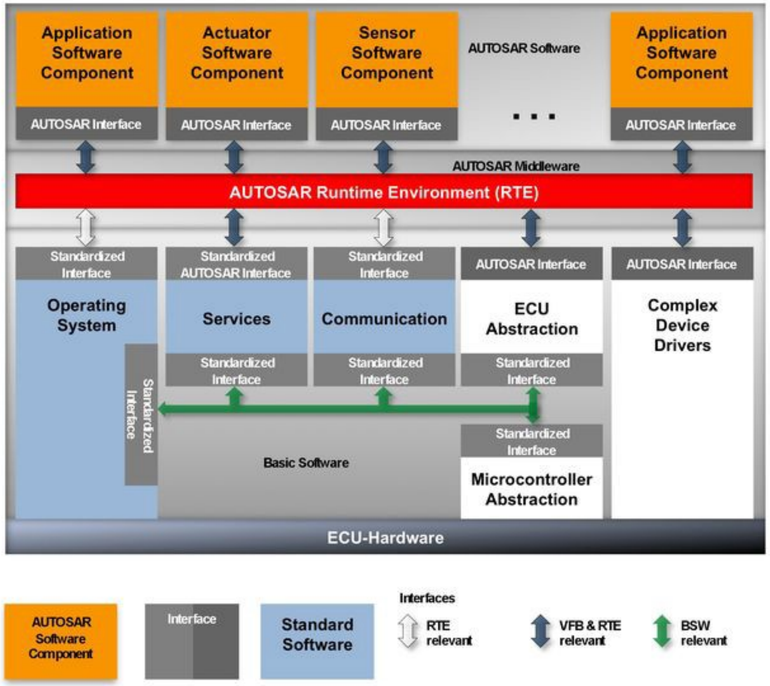
Partners & Funding

- Halmstad University
  Research in model-based testing and software verification
- Quviq A.B., Sweden
  Model-based testing tool QuickCheck, AUTOSAR models and testing expertise
- ArcCore A.B., Sweden
  AUTOSAR development environment, open source AUTOSAR implementation
- Funded by

Knowledge Foundation

AUTOSAR

- A comprehensive standard for building automotive software
- In particular, description of basic software components / libraries
- ~3k pages of text
- Examples:
  CAN-bus stack, FlexRay stack, memory access interfaces, hardware abstraction (e.g. PWM / ADC), …



Motivation

- Automotive Open System Architecture – AUTOSAR
- To enable pluggable components and multiple vendors
- Room for interpretation and optimisation
  - Intentional and inadvertent specification loopholes
  - Specific implementations differ (from each other and from the spec)
- Results in non-conformant components
- Can lead to serious problems in integration
- Research question – measure the severity, find the consequences

Goals

In the context of the AUTOSAR standard:

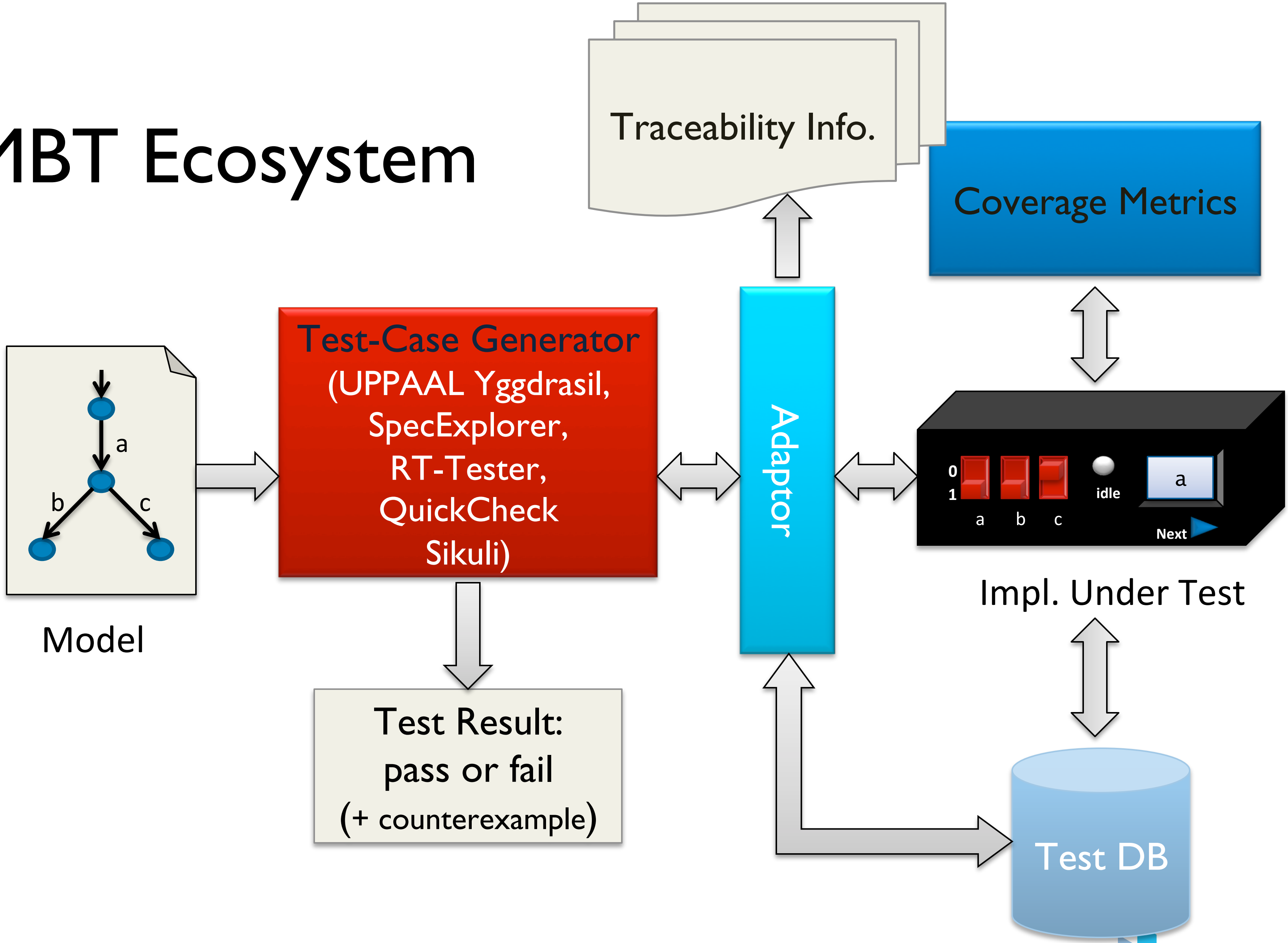
1 Measure the severity of deviations in non-conformant components; show how a selection in a given (complex) system leads to a failure (bottom-up)

2 Given a failure of the system and the knowledge of deviations in components, identify the root cause (top-down)

Means

1. Model-Based Testing (MBT)
2. Machine learning techniques
3. Symbolic execution

MBT Ecosystem



Traceability Info.

Coverage Metrics

Model

Test-Case Generator
(UPPAAL Yggdrasil,
SpecExplorer,
RT-Tester,
QuickCheck
Sikuli)

Adaptor

Impl. Under Test

Test Result:
pass or fail
(+ counterexample)

Test DB

Model Based Testing with QuickCheck

- Erlang based tool for guided random test generation Based
- on a state-full model / specification
- Can test functions in separation, but also their interaction
- Very snappy and cool! 🙂
- Probably more about this in John's talk

The First Task

1. Detect and classify non-conformances
2. Summarise / formalise them

The First Task

1. Detect and classify non-conformances
2. Generalize and summarise them

- Problem 1 is relatively easy:
  - Use QuickCheck and AUTOSAR models to find concrete failures

- Part 2 is to quickly detect whether a particular behaviour observed later falls into the non-conformance, a formal description of sorts

Specification of Non-Conformance

- Negative model of the component
- I.e. a description of what the non-conformance does
- Saturated to only that behaviour,
  other (correct) behaviours not in scope
- Can be parametric to further differentiate kinds of a particular non-conformance
- [What QuickCheck actually does for implementation variants]

Specification of Non-Conformance

- Negative model of the component
- I.e. a description of what the non-conformance does
- Saturated to only that behaviour,
  other (correct) behaviours not in scope
- Can be parametric to further differentiate kinds of a particular non-conformance
- [What QuickCheck actually does for implementation variants]

Question 1

How to generate it (semi-)automatically out of a (failing) test?

Constructing Negative Models

- Automata learning
- Normally used to learn the models of correct, black-box systems
- Now learn about failures / non-conformances
- Not so straightforward:
  - How can we be sure that we learn about one failure?
  - How to remove “noise” during learning?
  - How to keep the input alphabet small?
- LearnLib: Automata Learning framework implemented in Java (powerful and unfortunately complex)
- Interface LearnLib to QuickCheck

[S. Kunze et al., Generation of Failure Models through Automata Learning, WASA 2016]

Example

```
/* Given the requested size of a buffer, return
   the available space. */
size_t get_buffer_size(size_t req_size);

/* Return the pointer to the array. */
uint8* get_buffer_array();
```

Example

```
/*  Given the requested size of a buffer,  return
    the available space.  */
size_t get_buffer_size(size_t req_size);

/*  Return the pointer to the array.  */
uint8*  get_buffer_array();
```

What happens when:

- The requested size is 0 or negative?
- The available space is smaller than the requested size?
- The pointer?

  Or even…

hh.se

Example

```
/*  Given the requested size of a buffer,  return
    the available space.  */
size_t get_buffer_size(size_t req_size);

/*  Return the pointer to the array.  */
uint8*  get_buffer_array();
```

What happens when:

- The requested size is 0 or negative?
- The available space is smaller than the requested size?
- The pointer?
- Or even… what is actually returned in normal conditions? Requested size or available space?

Where is the Problem?

- Fine as long the surrounding environment is aware of the particular choice…

Where is the Problem?

- Fine as long the surrounding environment is aware of the particular choice…
- When intermixing implementations things will go bad!

hh.se

Where is the Problem?

- Fine as long the surrounding environment is aware of the particular choice…
- When intermixing implementations things will go bad!
- Typical problems:
  - Treatment of corner cases
  - Indexes and timing off by one
  - …

Symbolic Execution

- Run the program on symbols instead of concrete data
- "Split" the running on every decision point
- Collect the different execution paths
- Each path is defined by constraints over the program data
- Tricky bits are library function calls, iterations, and recursion

hh.se

Symbolic Execution Applications

- Popular in theorem proving / program logics for formal verification of programs
- Can be applied to the code or the model (QuickCheck models are executable)

hh.se

Symbolic Execution Applications

- Popular in theorem proving / program logics for formal verification of programs
- Can be applied to the code or the model (QuickCheck models are executable)
- Can be then used for Concolic Testing (**Con**crete / Symbo**lic**)
  - The set of execution paths provide test partitioning
  - Test data generated by constraint solving

Further Tasks

Question 2

Can a non-conformant component cause trouble?

Further Tasks

Question 2

Can a non-conformant component cause trouble?

```
/* Given the requested size of a buffer, return
   the available space. */
size_t get_buffer_size(size_t req_size);
```

Further Tasks

Question 2

Can a non-conformant component cause trouble?

```
/*  Given the requested size of a buffer,  return
   the available space.  */
size_t get_buffer_size(size_t req_size);
```

1. Return -1 when requesting too much
2. Return capacity when requesting too much

What Can Go Wrong?

```
if(get_buffer_size(128) <= 0) {
    /* Bail out / recover */
} else {
    /* Store 128 bytes of data in the buffer */
}
```

Behaviour 2 of get_buffer_size will cause a segmentation fault!

```
if(get_buffer_size(128) < 128) {
    /* Bail out / recover */
} else { ... }
```

Safe for both behaviours! How about other cases, especially generated software?

Further Tasks

Question 3

When the system fails / crashes – was it caused by a non-conformant component and if so, which one?

hh.se

Further Tasks

Question 3

When the system fails / crashes – was it caused by a non-conformant component and if so, which one?

First idea:

- Perform run-time checking of sorts
- Record traces of function calls and their parameters
- Check if they fall within the non-conformant model (specification) of any of the components
- Could be possibly done on a live system (ECU)

Conclusions

- Model-based testing: an effective method of bug hunting
- Bug fixing: a social process
- Demonstrating probability and severity of a bug facilitates the process:
    machine learning to generalize the failing test case
    symbolic execution to demonstrate bigger failures

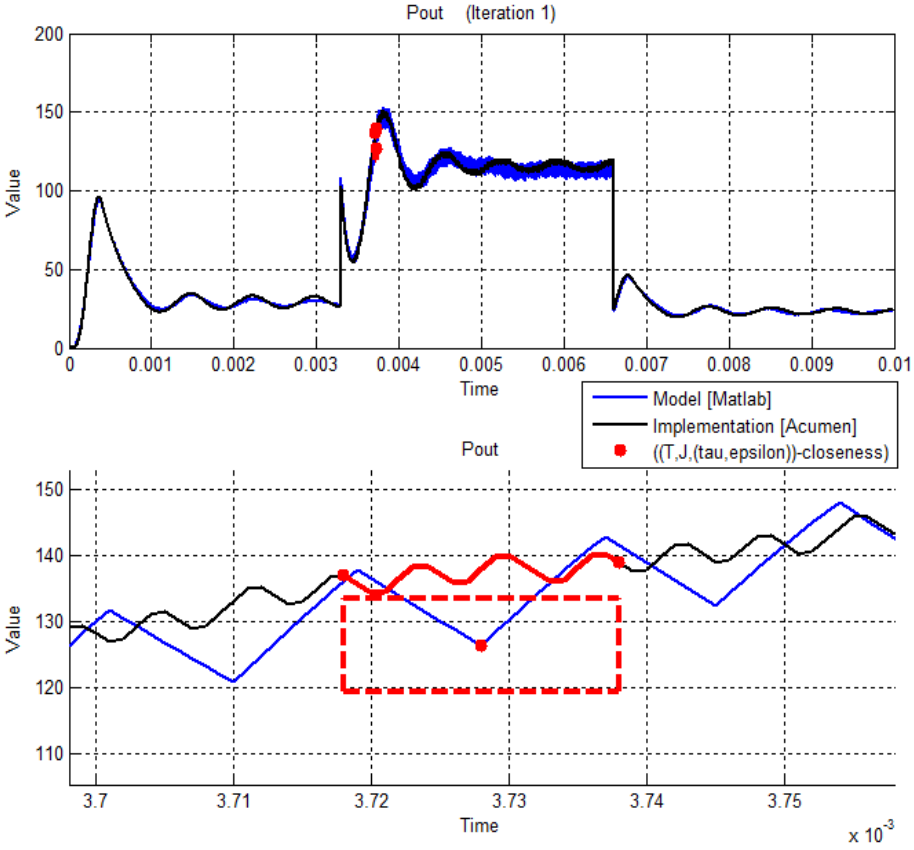
Next Steps

- Apply symbolic execution to search for consequences and to diagnose failures
- Apply to more realistic case studies (Arctic Studio implementations, fault injections)
- Implement necessary extensions in QuickCheck

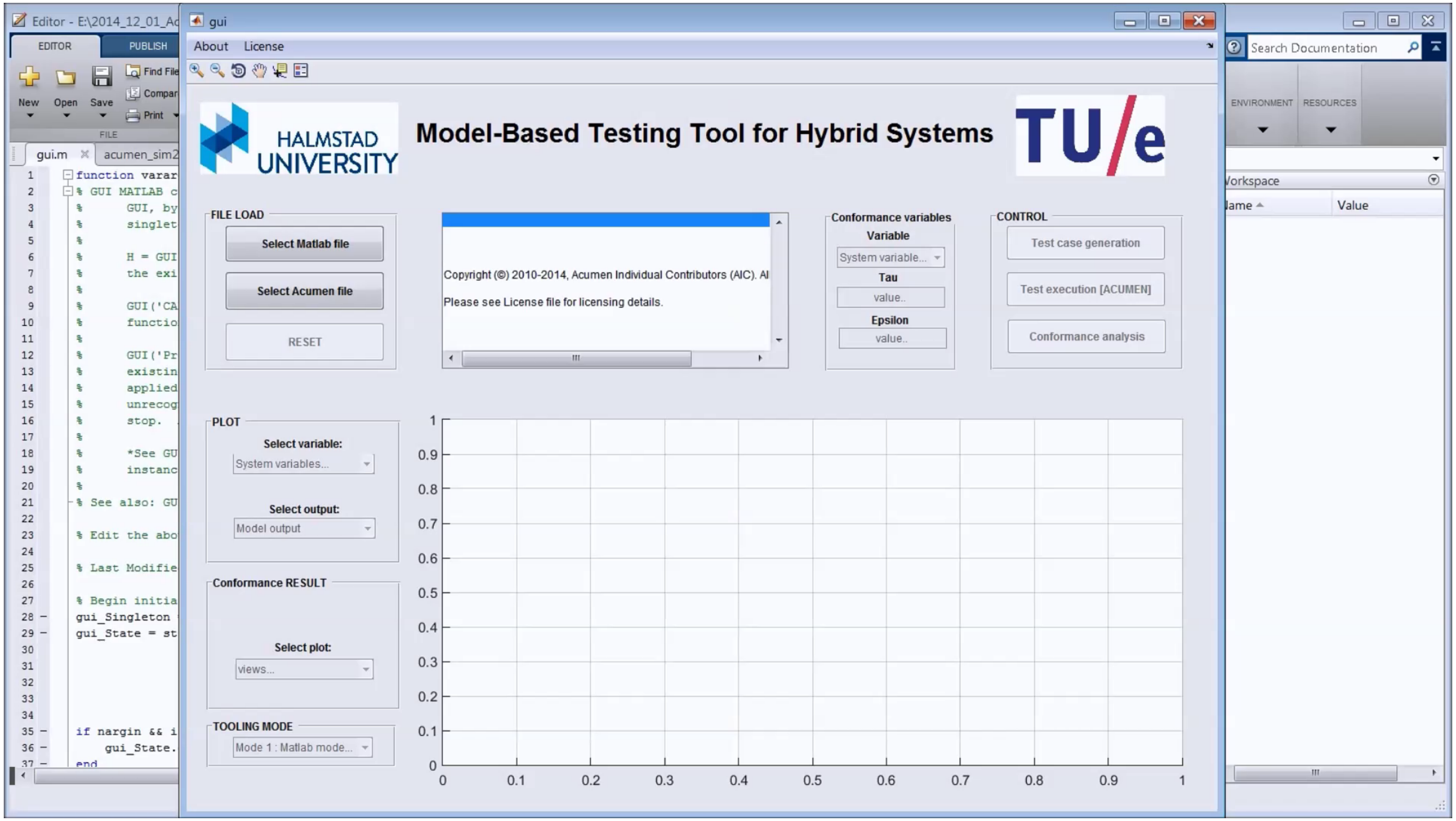
MBT for Cyber-Physical Systems

Challenges:

- Modeling system dynamics (differential equations, accuracy of numerics)
- Sampling inputs and outputs, approximate conformance (in time and value)
- Coverage



MBT for Cyber-Physical Systems



```
function varar
% GUI MATLAB c
%       GUI, by
%       singlet
%
%       H = GUI
%       the exi
%
%       GUI('CA
%       functio
%
%       GUI('Pr
%       existin
%       applied
%       unrecog
%       stop.
%
%       *See GU
%       instanc
%
% See also: GU

% Edit the abo

% Last Modifie

% Begin initia
gui_Singleton
gui_State = st




if nargin && i
    gui_State.
end
```

Copyright (©) 2010-2014, Acumen Individual Contributors (AIC). Al

Please see License file for licensing details.

[Aerts, Reniers, MRM.
Tool Prototype for Model-Based Testing of
Cyber-Physical Systems, ICTAC 2015]

6th Halmstad Summer School on Testing

http://ceres.hh.se/HSST_2016

Dino Distefano
FaceBook and
Queen Mary U.

Alastair Donaldson
Imperial College

Jeff Offutt
George Mason U.

Marielle Stoelinga
U. Twente

Alexandre Petrenko
Comp. Sys. Research Inst.

Per Runesson
Lund U.

HALMSTAD UNIVERSITY

Thank You!

Mohammad Mousavi
m.r.mousavi@hh.se
bit/ly/CERES_MBT