HALMSTAD UNIVERSITY

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

Wojciech Mostowski
Halmstad University, Sweden

AstaZero Researchers Day 2016

# Outline

# 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 specification)
- Results in non-conformant components
- Can lead to potentially serious problems in the software
- Research question – find the consequences

# Goals

In the context of the AUTOSAR standard:

1. Given a non-conformant set of components how can we show that there exists a selection in a given (complex) system that leads to a failure (bottom-up)

2. Given a failure of the system and the knowledge that non-conformant components were used, identify the one that is the root cause of the failure (top-down)

# Goals

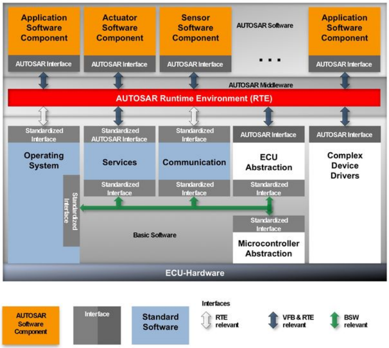
In the context of the AUTOSAR standard:

1. Given a non-conformant set of components how can we show that there exists a selection in a given (complex) system that leads to a failure (bottom-up)
2. Given a failure of the system and the knowledge that non-conformant components were used, identify the one that is the root cause of the failure (top-down)

Using Model-Based Testing (MBT) techniques

# 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), …



# 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

# 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_t* 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_t* 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…

# 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_t* 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!

# 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
  - …

# 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 interacting
- Hundreds of tests are generated and executed, minimal counter examples reported for the failed ones
- Very snappy 🙂

# QuickCheck Model – Queue of Integers

```
-record(state, {ptr, size, elements}).
initial_state() -> #state{ elements=[] }.
```



HALMSTAD UNIVERSITY

# QuickCheck Model – Queue of Integers

```
-record(state, {ptr, size, elements}).
initial_state() -> #state{ elements=[] }.
...
put_pre(S, [_P, _E]) -> S#state.ptr /= undefined andalso
    length(S#state.elements) < S#state.size.
put_next(S, _R, [_P, E]) ->
    S#state{ elements = S#state.elements ++ [E] }.
put_post(_S, [_P, E], R) -> R == E.
```

# QuickCheck Model – Queue of Integers

```
-record(state, {ptr, size, elements}).
initial_state() -> #state{ elements=[] }.
...
put_pre(S, [_P, _E]) -> S#state.ptr /= undefined andalso
    length(S#state.elements) < S#state.size.
put_next(S, _R, [_P, E]) ->
    S#state{ elements = S#state.elements ++ [E] }.
put_post(_S, [_P, E], R) -> R == E.
...
prop_q() -> ?FORALL(Cmds, commands(?MODULE),
     begin
       {H, S, Res} = run_commands(?MODULE, Cmds),
       collect(S, pretty_commands(?MODULE, Cmds,
           {H, S, Res}, Res == ok))
     end).
```

# AUTOSAR Models by QuviQ

- Multiplicity of models for basic AUTOSAR software
- Implementations of clients tested for conformance
- Bugs found (obviously), but also problems with the specification
- Base for the work ahead of us

| LIN | CAN | FlexRay | Ethernet |
| --- | --- | --- | --- |
| LinNm | | | EthSa |
| LinSm | CanNm | FxNm | EthNm |
| LinIf | CanSm | FxSm | EthSm |
| LinTrcv | CanTp | FxTp | EthIf |
| Lin | CanIf | FxIf | |

# First Steps in the Project

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

# First Steps in the Project

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

- Problem 1 is relatively easy:
  - Use QuickCheck and AUTOSAR models to find failures
  - Verify them (manually) to be a non-conformance of an implementation (rather than a problem in the specification or model)



HALMSTAD UNIVERSITY

# First Steps in the Project

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

- Problem 1 is relatively easy:
  - Use QuickCheck and AUTOSAR models to find failures
  - Verify them (manually) to be a non-conformance of an implementation (rather than a problem in the specification or model)

- Part 2 is about learning something more about the non-conformance
  - Failed test gives only one counter example
  - What are the other failing behaviours? How can they be described?

# Failure Models

- State-full specification showing under which circumstances / execution traces a component will lead to a failure
- Build from the information about single counter examples
- Through the automata learning process
- The result is a Mealy machine – automata with inputs and outputs:
  - Inputs are abstracted concrete inputs of the system under test
  - Outputs are the success / failure of the test so far
  - States represent the states of the correct behaviour plus one failure state

# Failure Models

- State-full specification showing under which circumstances / execution traces a component will lead to a failure
- Build from the information about single counter examples
- Through the automata learning process
- The result is a Mealy machine – automata with inputs and outputs:
  - Inputs are abstracted concrete inputs of the system under test
  - Outputs are the success / failure of the test so far
  - States represent the states of the correct behaviour plus one failure state

**Challenge**

Devise this process so that it is feasible and the result is readable



HALMSTAD UNIVERSITY

# Failure Model Learning Process

## A bridge / interface

Mediate between the test running in QuickCheck and the automata learning framework LearnLib

# Failure Model Learning Process

## A bridge / interface

Mediate between the test running in QuickCheck and the automata learning framework LearnLib

## User guidance

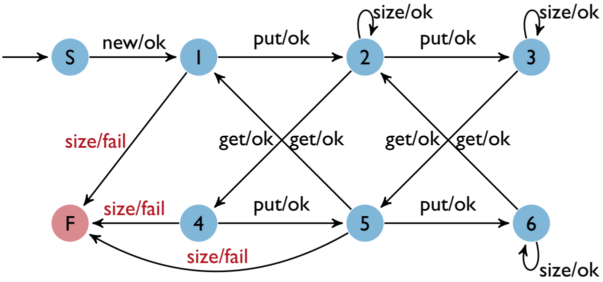
- Which concrete parameters of the SUT can be randomly generated, which have to be fixed
- So that the model is concise and learned in reasonable time
- That is, without guidance there might be too much to learn

# Example

## Learning a Faulty Queue Implementation

- The new operation that initialises the queue should always use the same size. Learning about queues of all arbitrary sizes in one go is not feasible.
- The put operation can use random parameters. Elements stored in the queue are not part of the model.

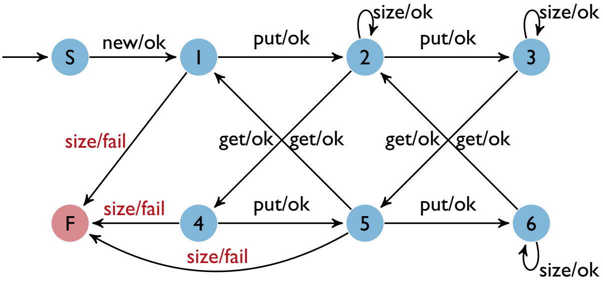
# Example

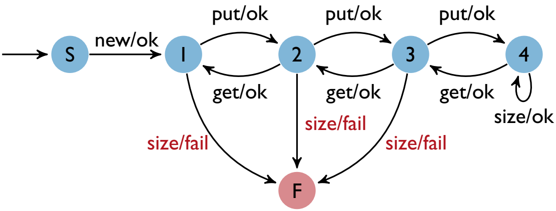





HALMSTAD UNIVERSITY

# Example


S
new/ok
I
put/ok
2
size/ok
put/ok
3
size/ok
size/fail
get/ok
get/ok
get/ok
get/ok
F
size/fail
4
put/ok
5
put/ok
6
size/fail
size/ok


S
new/ok
I
put/ok
2
put/ok
3
put/ok
4
get/ok
get/ok
get/ok
size/ok
size/fail
size/fail
size/fail
F

# Summary

- First phase of the project, fault learning methods
- Using toy examples
- Working prototype of the fault learner
- Apply to more realistic case studies
  (Arctic Studio implementations, fault injections)
- Use failure models for fault consequence analysis

# Summary

- First phase of the project, fault learning methods
- Using toy examples
- Working prototype of the fault learner
- Apply to more realistic case studies
  (Arctic Studio implementations, fault injections)
- Use failure models for fault consequence analysis

# Thank You!



HALMSTAD UNIVERSITY