HALMSTAD UNIVERSITY

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

Wojciech Mostowski
Mohammad R. Mousavi

Halmstad University, Sweden

Outline

1. Project overview
2. Consortium
3. Model-based testing of AUTOSAR
4. Fault model learning
5. Conclusion

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 problems in the software
- Research question – find the consequences

Goals

In the context of the AUTOSAR standard:

- Given a set of components that may be non-conformant how can we show that there is a combination of them that leads to a failure (bottom-up)

Goals

In the context of the AUTOSAR standard:

1. Given a set of components that may be non-conformant how can we show that there is a combination of them that leads to a failure (bottom-up)

   What is the limit of freedom in making component implementation choices for a given application and safety requirements?

2. Given a failure of the system, identify the component(s) that were the root cause of the failure (top-down)

Goals

In the context of the AUTOSAR standard:

1. Given a set of components that may be non-conformant how can we show that there is a combination of them that leads to a failure (bottom-up)

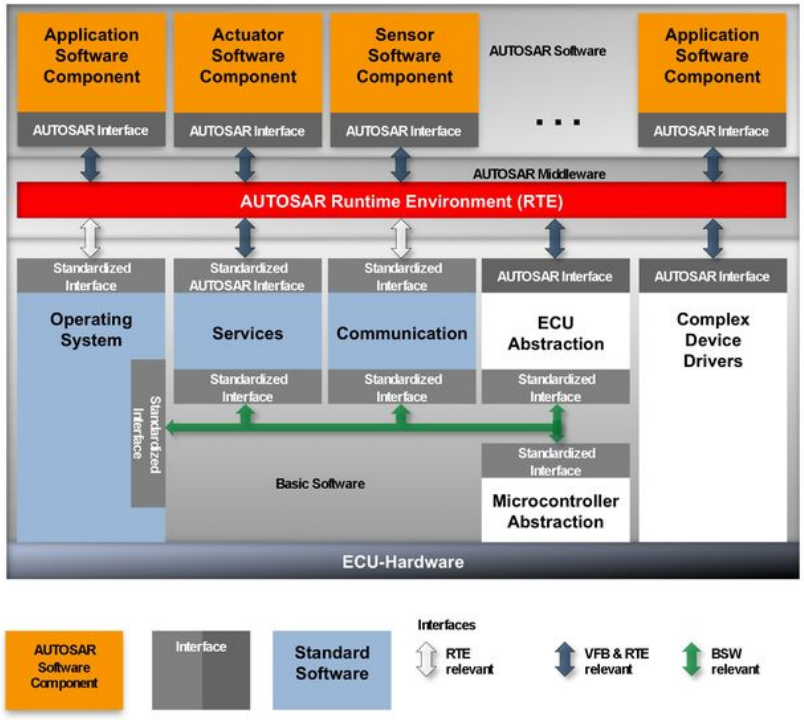
   What is the limit of freedom in making component implementation choices for a given application and safety requirements?

2. Given a failure of the system, identify the component(s) that were 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
- Thousands of pages of text
- Examples:
  CAN-bus stack, FlexRay stack, memory access interfaces, hardware abstraction (e.g. PWM / ADC), …



HALMSTAD UNIVERSITY

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

HALMSTAD
UNIVERSITY

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?

- A particular implementation choice is fine if the client system accounts for it:
  - The code is tailored to one particular behaviour
  - And tested in this one context

Where is the Problem?

- A particular implementation choice is fine if the client system accounts for it:
  - The code is tailored to one particular behaviour
  - And tested in this one context
- Re-plugging one component may or may not break the system!

Where is the Problem?

- A particular implementation choice is fine if the client system accounts for it:
  - The code is tailored to one particular behaviour
  - And tested in this one context
- Re-plugging one component may or may not break the system!
- How to test it without developing several big test suites?
- Typical problems:
  - Treatment of corner cases
  - Indexes or 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
- Quick and automatic given model availability

Property-Based Specifications

- Every function is guarded by a precondition – specification of allowable calls, e.g., parameter range, the state of the model
- And a postcondition – property of the state and return value of the function to be checked

Property-Based Specifications

- Every function is guarded by a precondition – specification of allowable calls, e.g., parameter range, the state of the model
- And a postcondition – property of the state and return value of the function to be checked
- A callout specification, requirement on the underlying calls, their presence or absence, sequencing, parameter range, etc.

Mocking

The last part is achieved by mocking parts of the system under test – replacing the implementations with their models, which are executable! This in turn enables call tracing.

The QuickCheck Testing Process

- The model is analysed and valid execution traces are randomly generated
- Similarly, random input data is generated according to specification
- Generated traces are executed on SUT
- Outputs are validated (postconditions & callouts)
- In case of failures, the counter example is printed, and an attempt to shrink it is made

QuickCheck Model – Queue of Integers

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

QuickCheck Model – Queue of Integers

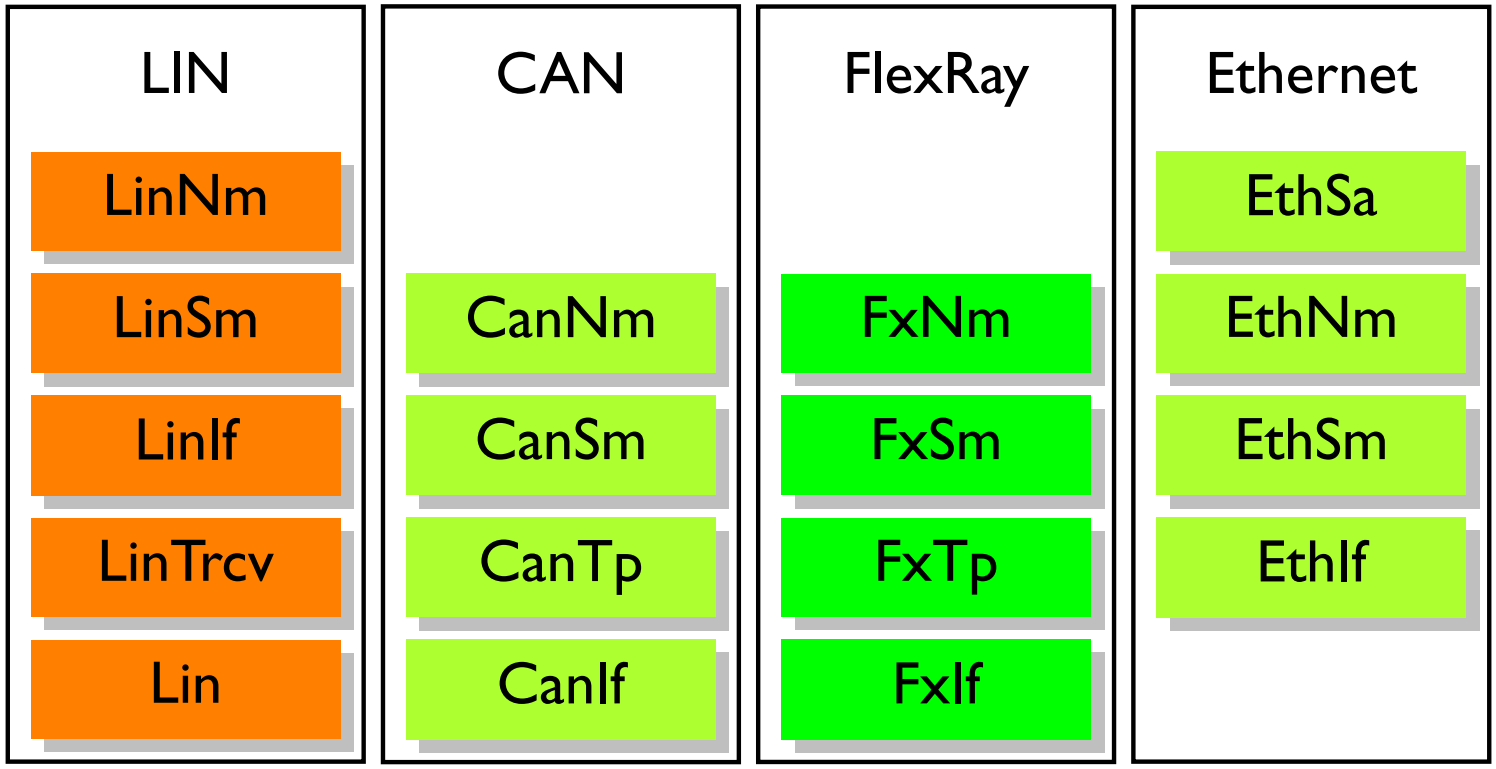
```
-record(state, {ptr, size, elements}).
initial_state() -> #state{ elements=[] }.
...
put_pre(S, [_P, _E]) -> S#state.ptr /= undefined andalso
    length(S#state.elements) < S#state.size.
put_next(S, _R, [_P, E]) ->
    S#state{ elements = S#state.elements ++ [E] }.
put_post(_S, [_P, E], R) -> R == 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
- AUTOSAR implementations tested for conformance
- Bugs found (obviously), but also problems with the specification

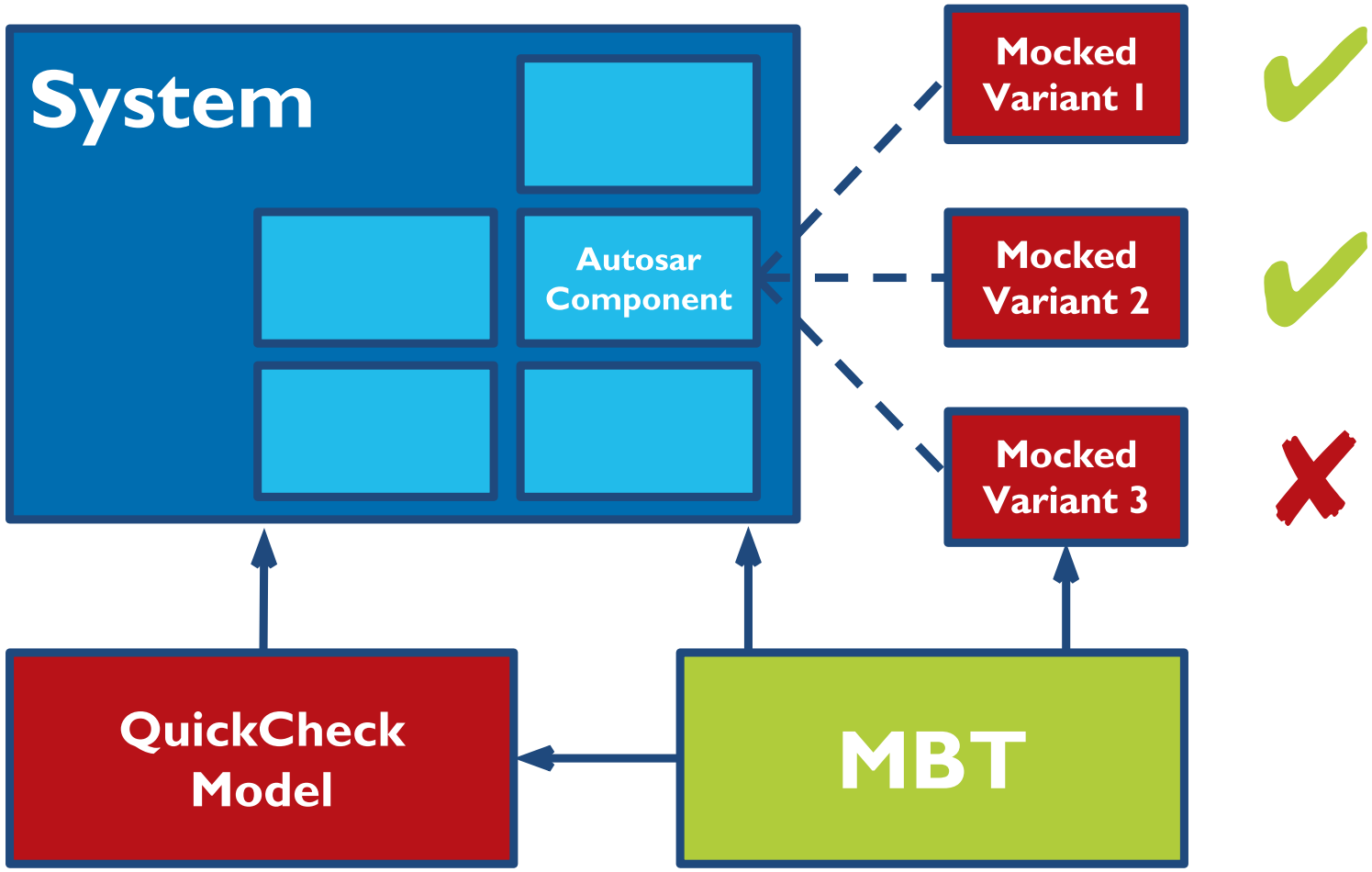


HALMSTAD UNIVERSITY

Consequence Testing

- Mock a basic software component
- Mocked component: correct or faulty, based on fault injection or known non-conformance
- Mocking by executing the model that accounts for non-conformances
- Specify and Model-Based-Test the complete system to check if that causes failures
- Similar process can be used to test for high-level safety properties, c.f. ISO-26262:
  - Safety properties specified as sequenced events in the system
  - "Input call X always triggers output call Y"

Consequence Testing



HALMSTAD UNIVERSITY

How to Specify a Fault?

- Mocked component: correct or faulty, based on fault injection or known non-conformance
- Mocking by executing the model that accounts for non-conformances

How to Specify a Fault?

- Mocked component: correct or faulty, based on fault injection or known non-conformance
- Mocking by executing the model that accounts for non-conformances

Failure Models

- The model needs to know about the faults
- When testing for faults, only single execution traces are reported, not the overall failure behaviour

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

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

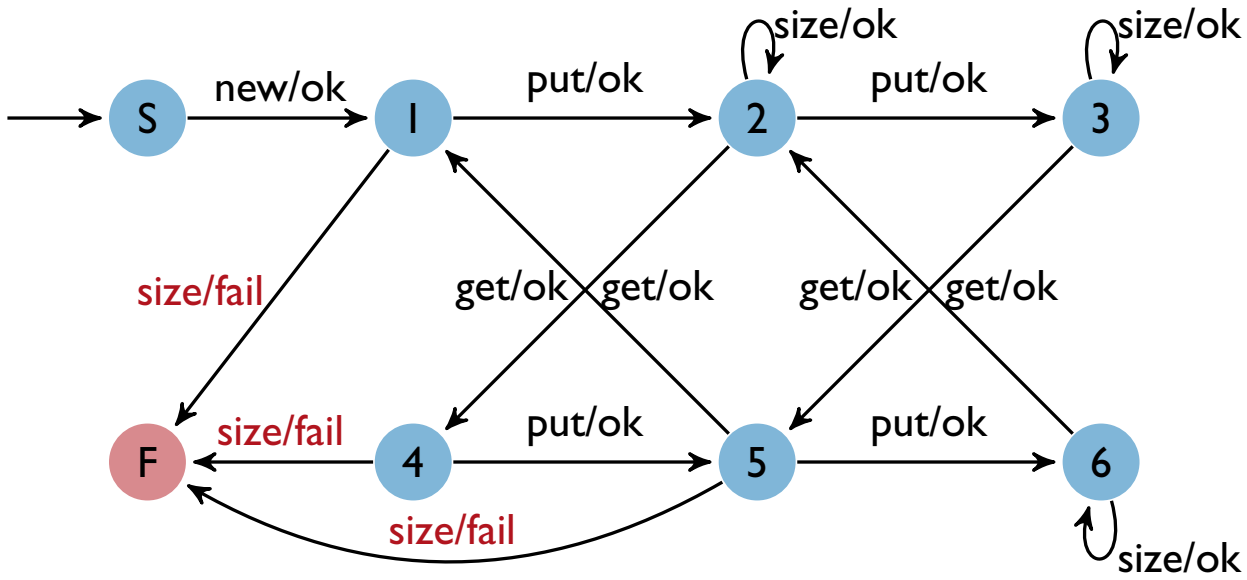
- The **new** operation that initialises the queue should always use the same size.

  Learning about queues of all arbitrary sizes in one go is not feasible.

- The **put** operation can use random parameters.

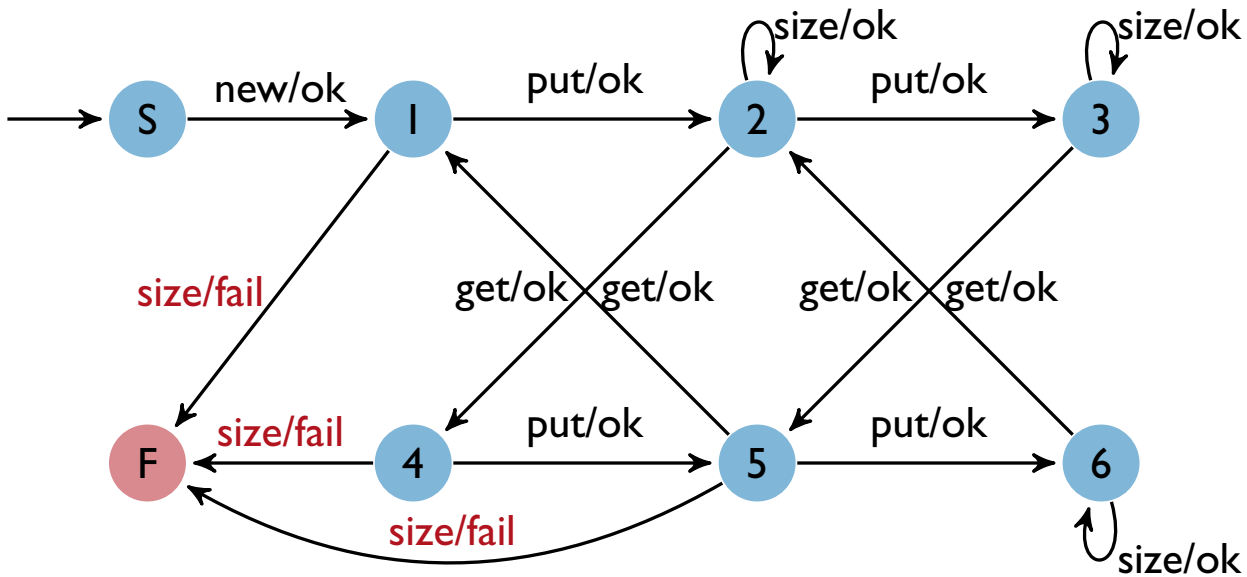
  Elements stored in the queue are not part of the model.

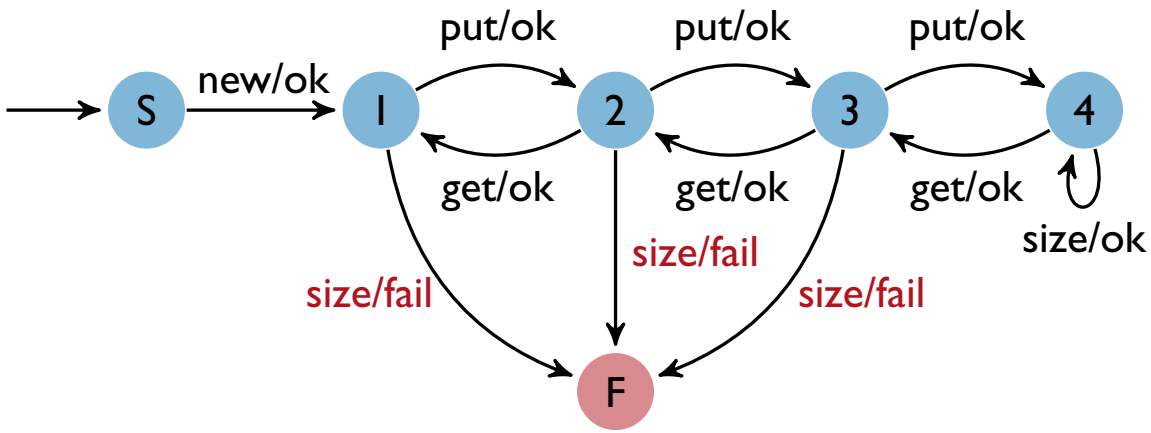
Example



HALMSTAD UNIVERSITY

Example





HALMSTAD UNIVERSITY

Summary

- Testing of failure consequences using Model-Based Testing and QuickCheck
- Failures need to be formalised – failure models
- Derived using automata learning
- Working prototype of the fault learner
- Application to more realistic case studies in progress (ArcCore's open-source AUTOSAR implementation)

Summary

- Testing of failure consequences using Model-Based Testing and QuickCheck
- Failures need to be formalised – failure models
- Derived using automata learning
- Working prototype of the fault learner
- Application to more realistic case studies in progress (ArcCore's open-source AUTOSAR implementation)

Thank You!