QuviQ

Testing Automotive Software with Erlang

Thomas Arts

Chalmers / Quviq AB

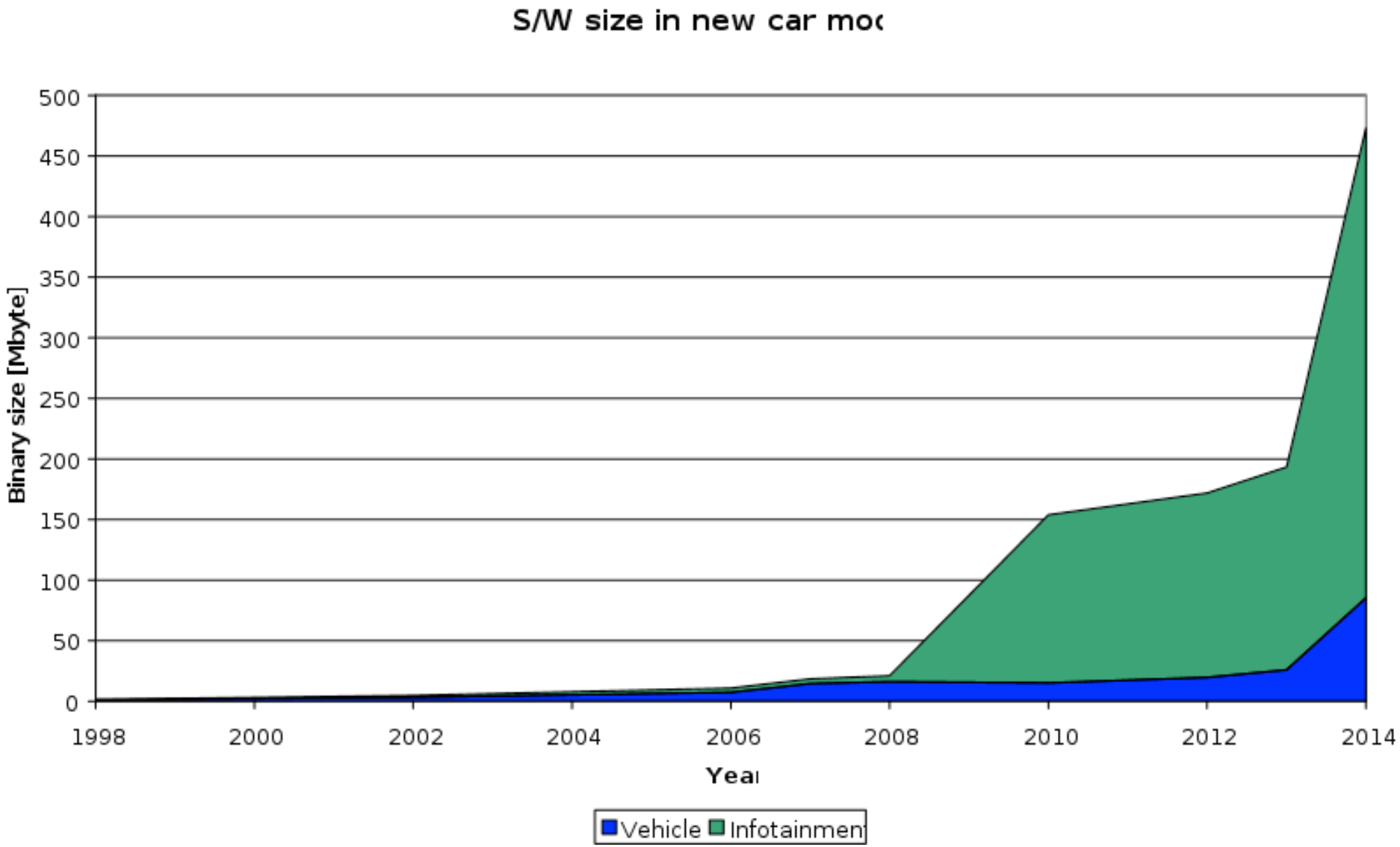
in collaboration with

Juan Puig, Anders Kallerdahl and Ulf Norell

Erlang Solutions Mentor Graphics Quviq

ProTest
property based testing

Software in modern cars



source: Ulrik Eklund

Many components that need to communicate with each other

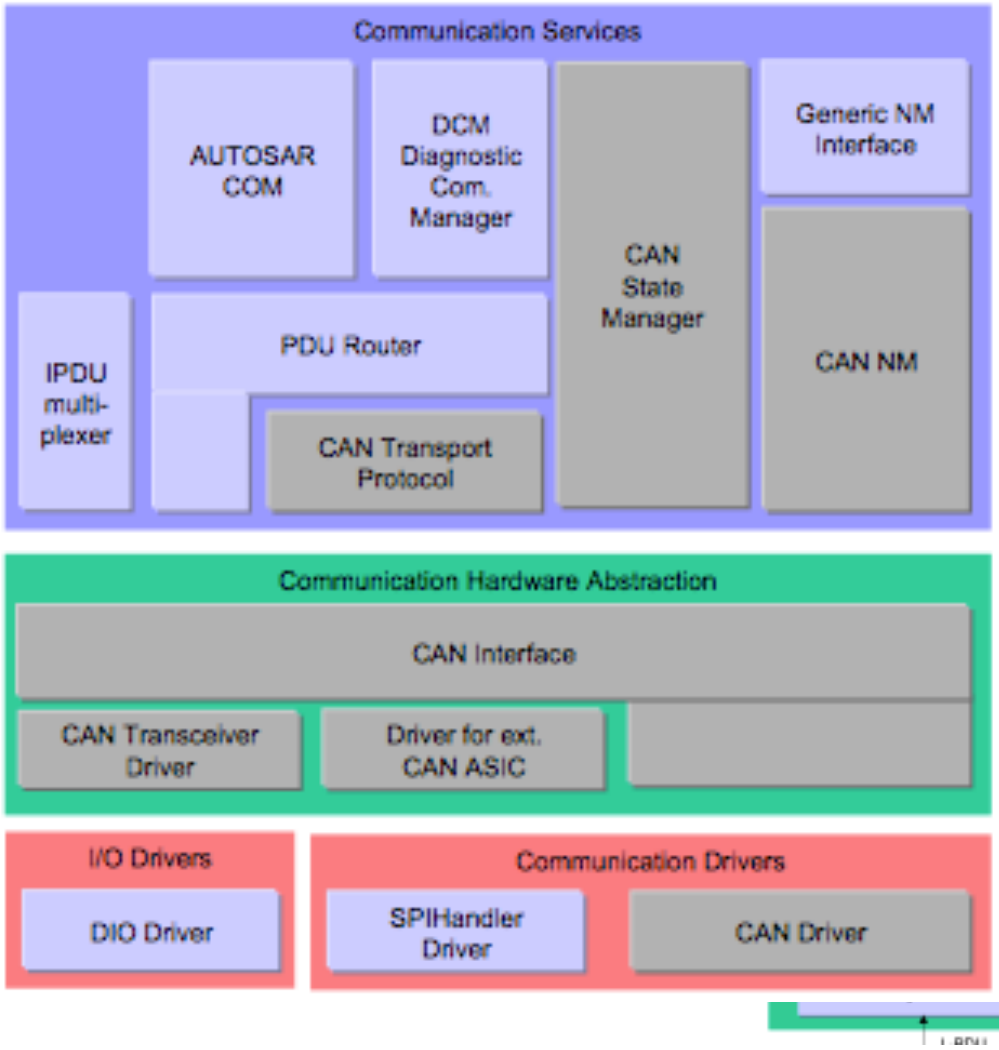
More diversity, faster time to market, higher complexity....

We have seen this before ☺

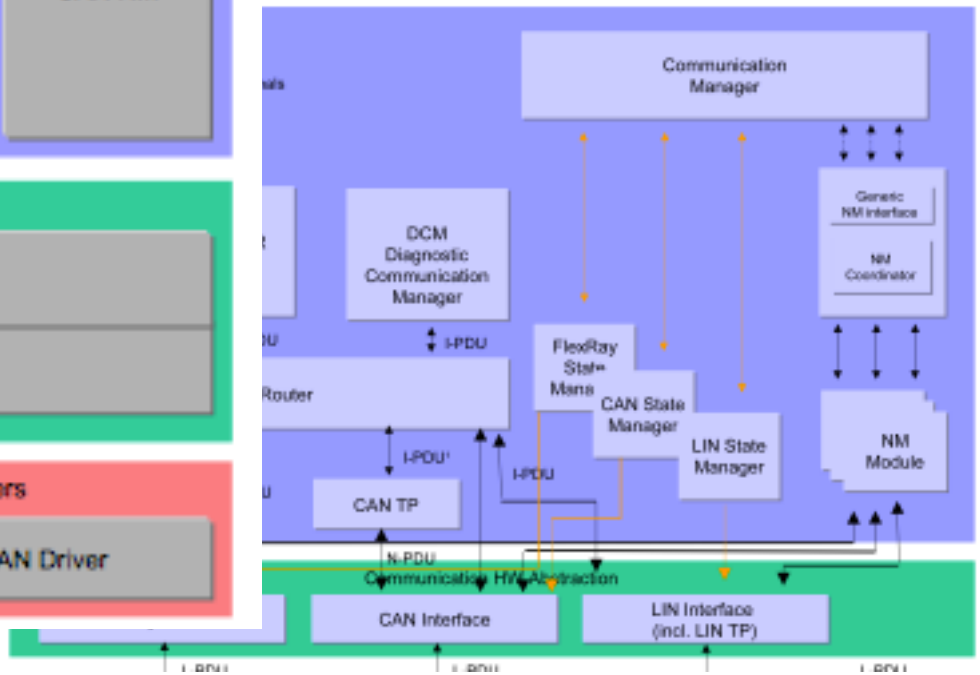
Solutions:

- Standardization of components
- Standard platform (operating system)

AUTOSAR a consortium standard



source: www.autosar.org



AUTOSAR specification open for interpretation.

Even if a component follows the standard, there is no guarantee at all that it will work in combination with other standard components

nothing new... we have seen that before ☺

The evil is hidden in configurations: each Node in the car has typically its own set of options, and software supplier

AutoSAR specification open for interpretation.

Even if a component follows the standard, there is no guarantee at all that it will work in combination with other stan

Solution:
Spend your budget on testing instead of development

nothing new... we have seen t

The evil is hidden in configura

the car has typically its own set of options, and software supplier

Software systems more complex every day...

... more components

... more possible configurations per component

... more component interactions

Traditional testing insufficient to keep up with this

We need to change our testing methods!

Using Erlang to test C software

- High level language: easier to write test code
- Good tools to support testing

but... we need to connect to C code

All information you need to write marshalling code is in the C (header) files.

Thus, we wrote a C parser in Erlang, extract all type information and generate the link between C and Erlang.

Suppose we have C file `example.c`

```
// Sum an array of integers
int sum (int *array, int len) {
  int n;
  int sum = 0;
  for (n = 0; n < len; n++)
     sum += array[n];
  return sum;
}
```

Erlang shell used to communicate with C

```
1> eqc_c:start(example).
ok

2> P = eqc_c:create_array(int, [1, 3, 3, 8]).
{ptr,int,1048864}

3> example:sum(P, 4).
15

4> eqc_c:free(P).
ok
```

Erlang shell used to communicate with C

```
1> eqc_c:start(example).
ok

2> P = eqc_c:create_array(int,
{ptr,int,1048864}

3> example:sum(P, 4).
15

4> eqc_c:free(P).
ok
```

- parse example.c
- create a c program that listens to a socket
- create example.beam and example.hrl with all functions from example.C
- start C program in a separate thread

Erlang shell used to communicate with C

```
1> eqc_c:start(example).
ok

2> P = eqc_c:create_array(int, [1, 3, 3, 8]).
{ptr,int,1048864}

3> example:sum(P, 4).
15

4> eqc_c:free(P).
ok
```

an array is created in the C thread and the pointer returned points to memory in that thread

From test case to property

Instead of specifying one or two test cases to demonstrate that the software fulfills a certain property, we specify ***the property*** and have the tests automatically generated!

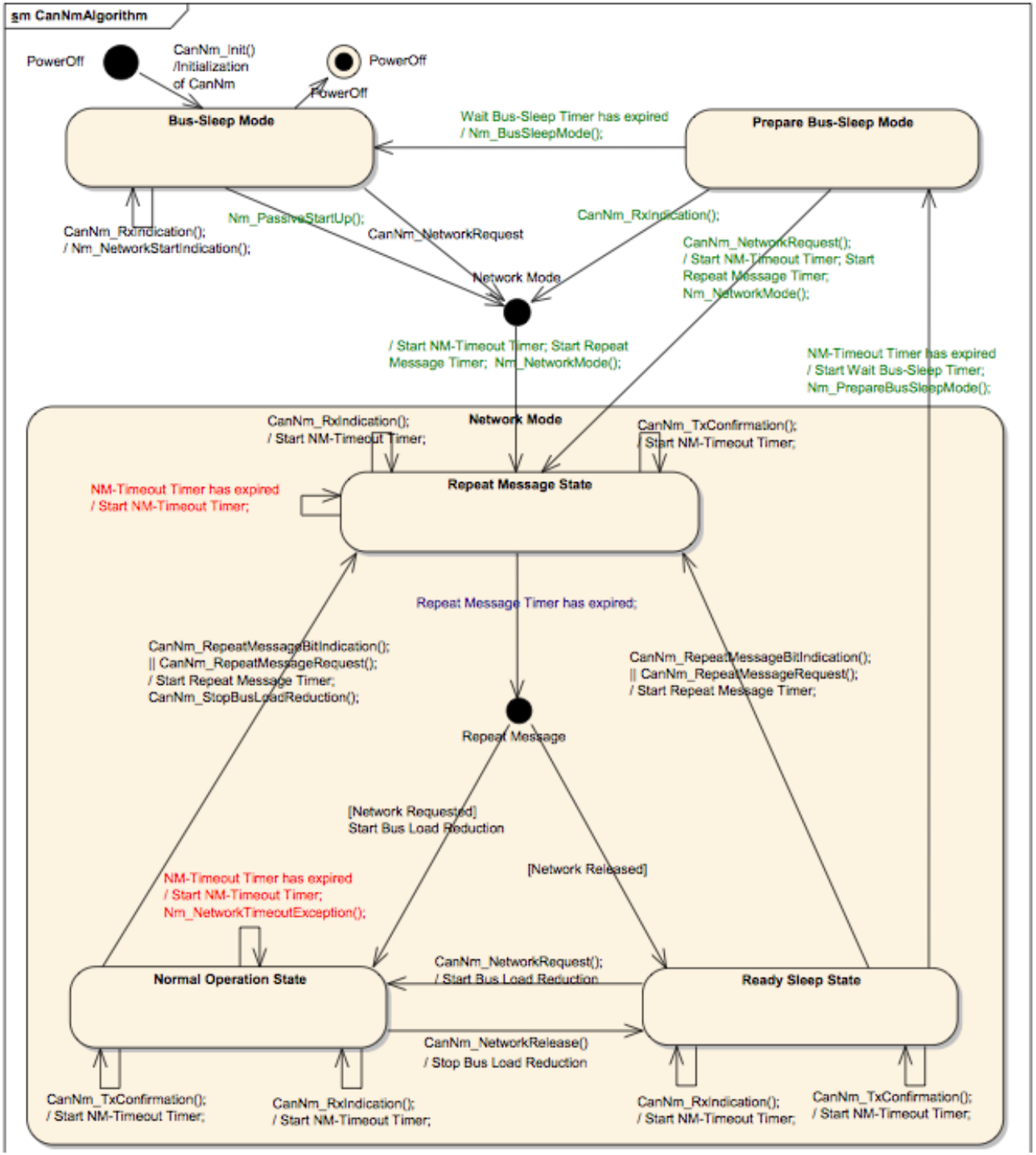
Model based testing with *controlled random generation of test cases*

How difficult is it to test real-time C code?

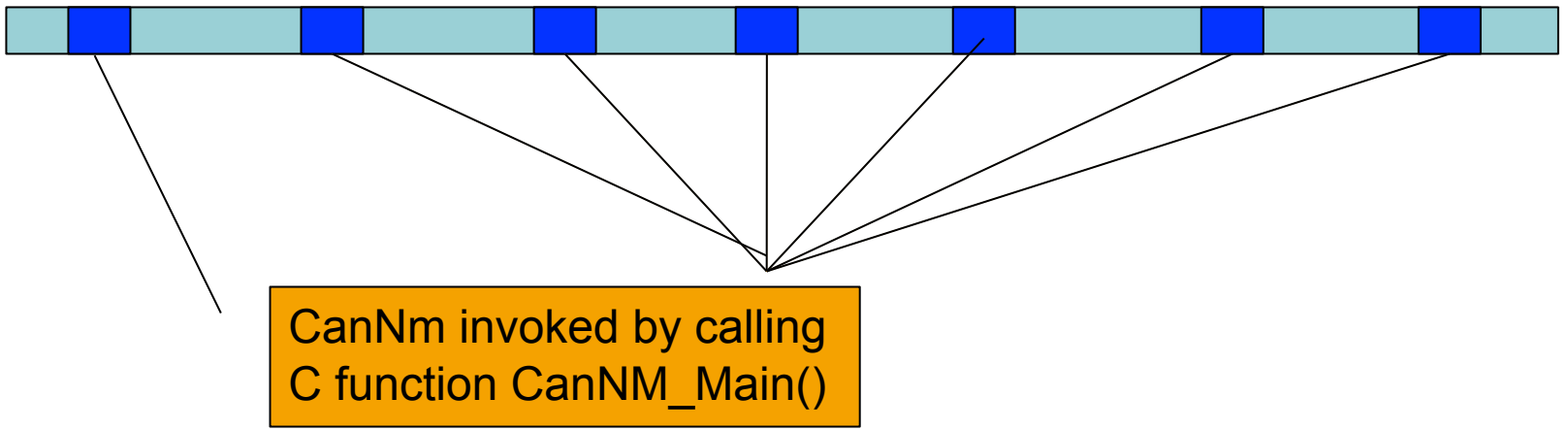
Mentor Graphics hosts master student thesis project to test CanNM with QuickCheck using this C link.

AUTOSAR component
as UML state machine

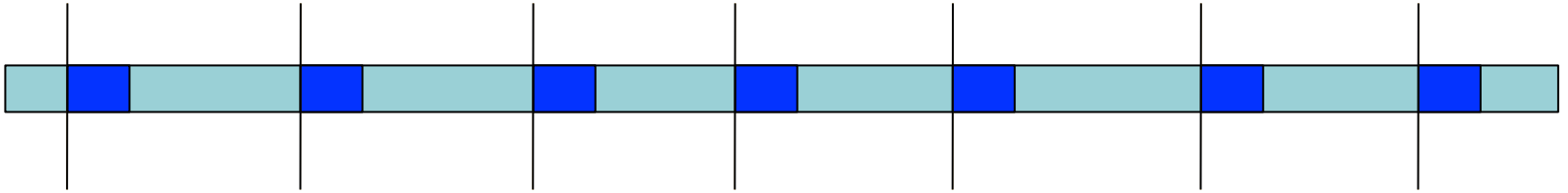
CAN
Network Management



CanNM is scheduled as one of many tasks



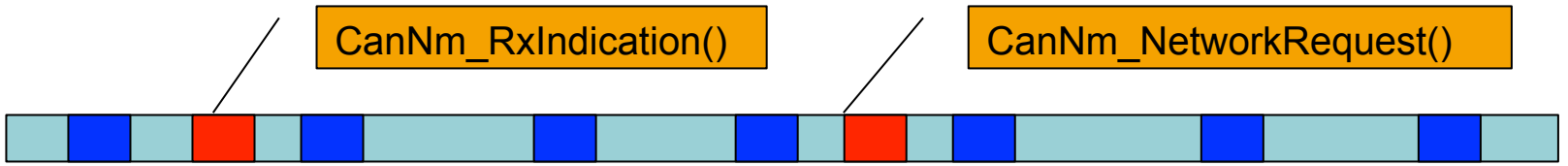
CanNM is scheduled as one of many tasks



Assumption:
One time unit elapses before CanNm_Main() is called

(In fact, C implementation handles the timers, not the scheduler)

CanNM is scheduled as one of many tasks



Other tasks communicate by calling CanNM interface functions

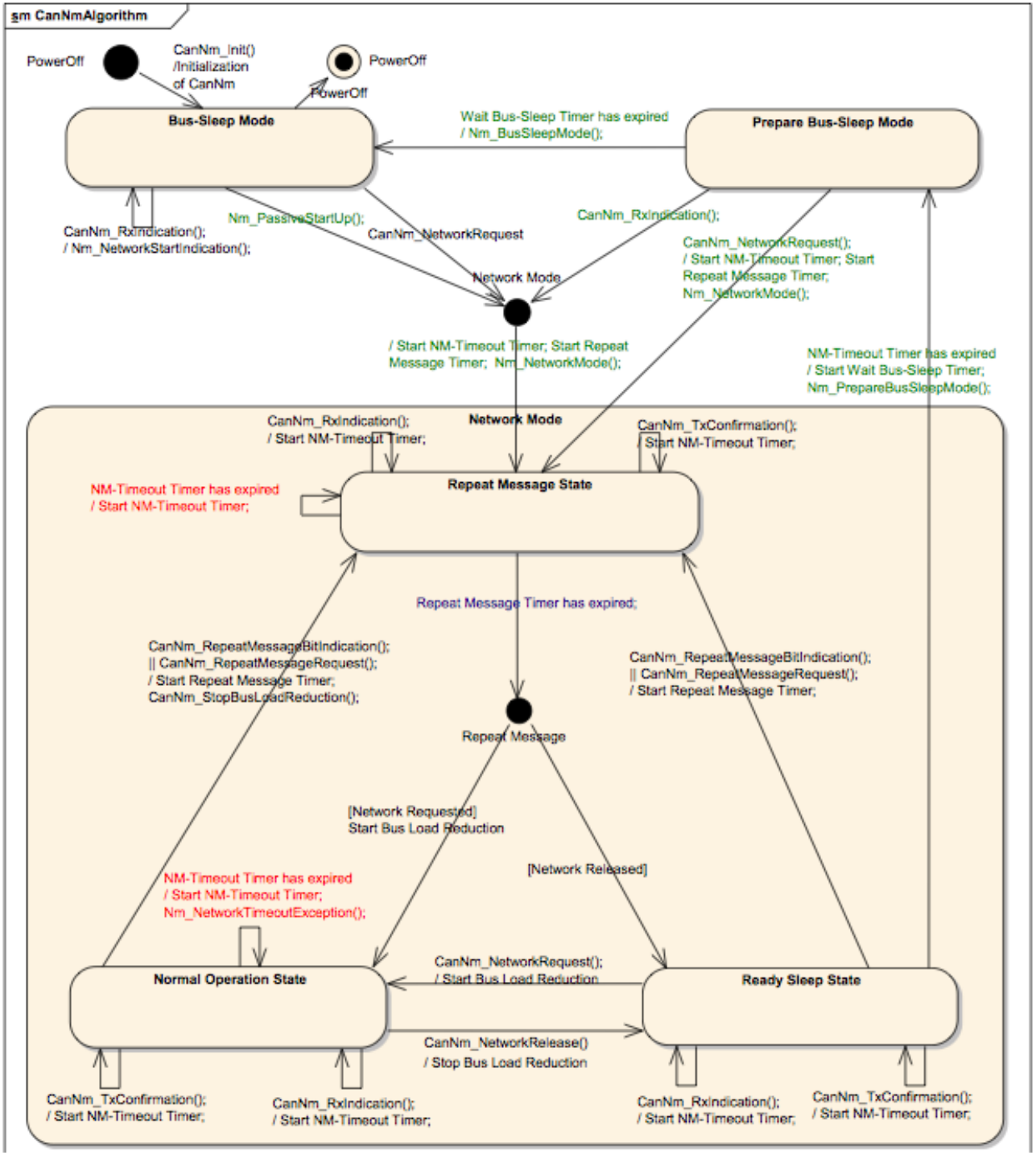
These update data structures in memory

Assumption: Only one interaction in each slot

AUTOSAR component as UML state machine

CAN
Network Management

Now... make a QuickCheck model from this state machine



State transitions as Erlang data structure

```
bus_sleep_mode(_) ->
    [ {power_off,{call,?MODULE,powerOff,[]}},
      {bus_sleep_mode,{call,?MODULE,main,[]}},
      {bus_sleep_mode,{call,?MODULE,'CanNm_RxIndication',[id(),u8()]}},
      {repeat_message_state,{call,?MODULE,'Nm_PassiveStartUp',[]}},
      {repeat_message_state,{call,?MODULE,'CanNm_NetworkRequest',[]}}].

repeat_message_state(_) ->
    [ {normal_operation_state,{call,?MODULE,main,[]}},
      {ready_sleep_state,{call,?MODULE,main,[]}},
      {repeat_message_state,{call,?MODULE,main,[]}},
      {repeat_message_state,{call,?MODULE,'CanNm_RxIndication',[id(),u8()]}},
      {repeat_message_state,{call,?MODULE,'CanNm_TxConfirmation',[id()]}}].
```

QuickCheck model

Model how additional state data changes: timers, network status, ...

```
next_state_data(repeat_message_state,repeat_message_state,S,_V,{_,_,main,_}) ->
    S#can_nm{repeatMessageTimer = S#can_nm.repeatMessageTimer-1,
             nmTimeoutTimer =
               case S#can_nm.nmTimeoutTimer of
                   0 -> ?NMTIMEOUT;
                   N -> N-1
               end};

next_state_data(repeat_message_state,repeat_message_state,S,_V,{_,_,_,_}) ->
   S#can_nm{repeatMessageTimer = S#can_nm.repeatMessageTimer-1,
            nmTimeoutTimer = ?NMTIMEOUT};
```

How difficult is it to test real-time C code?

Master student thesis project to test CanNM with QuickCheck using this C link.

Result: - we know how to do it
- it is not that much work
- we found ambiguities in the specification

CanNm was modeled using a state machine.

Not all AUTOSAR components are specified as state machines... can we do the rest as well?

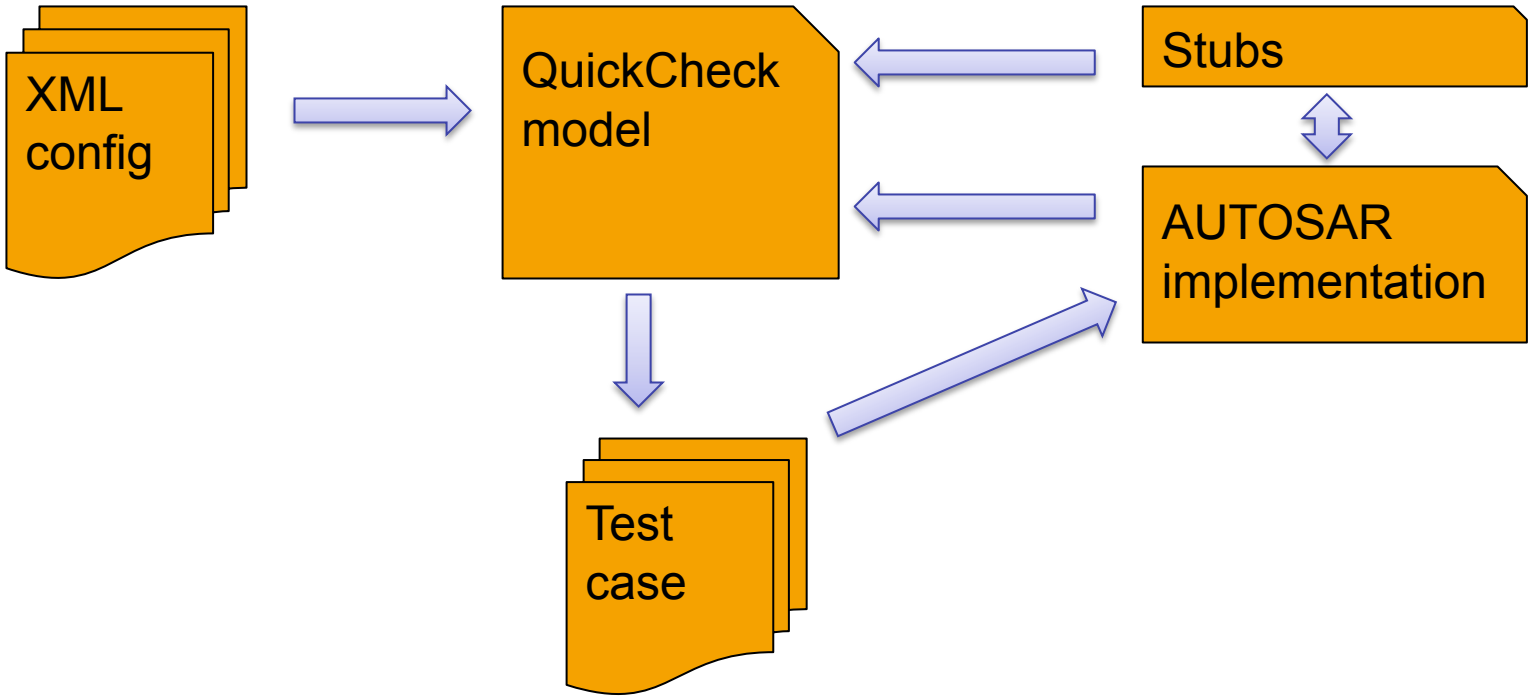
Sep/Oct 2010: Experiment (with Mentor Graphics)

- Test COM/PDUR with QuickCheck
- In parallel manual testing of same software (estimated 20 weeks)

approx 8000 lines of C code, representative component

Testing COM/PduRouter

- We have built a model for testing COM and PduRouter



We created a model

The model is configurable with an XML config file

Marshalling code is automatically generated from header files

C stub is only a 400 lines of code

QuickCheck model is 800 lines of code

Total: 2 person weeks work

Conclusions:

We gain productivity

- Erlang less lines of code
- QuickCheck model instead of test cases

We have a scalable solution for AUTOSAR

In the future...

buy a car that has been tested with Erlang!