Formal Methods in Practice: Tools and Examples

Alessio Ferrari[1]
alessio.ferrari@ge.com

[1]University of Florence, D.S.I., Florence, Italy

November 29, 2009

Outline

Why shall we use formal methods?

Because we do not want another Ariane 5 accident...

...in June 4, 1996, US $ 500 were blown away in 37 seconds for a software bug

...and that is only one of the many software-related accidents...

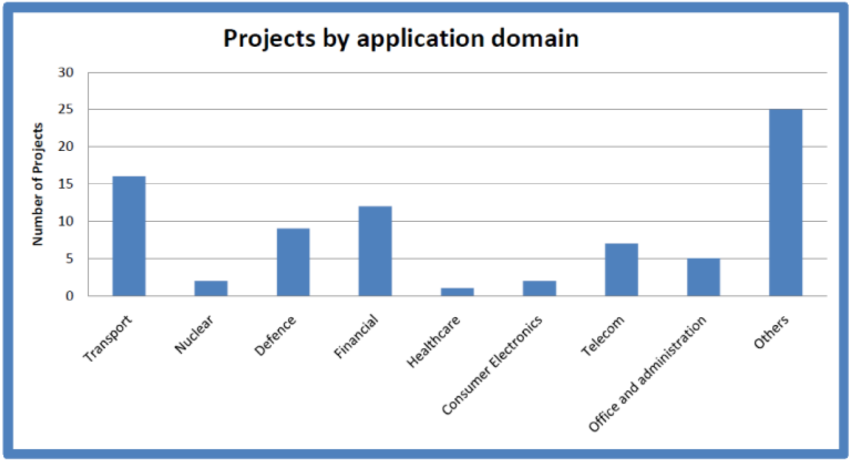
July 28, 1962 – Mariner I space probe: A bug in the flight software for the Mariner 1 causes the rocket to divert from its intended path on launch. Mission control destroys the rocket over the Atlantic Ocean. The investigation into the accident discovers that a formula written on paper in pencil was improperly transcribed into computer code, causing the computer to miscalculate the rocket's trajectory.
1985-1987 – Therac-25 medical accelerator: Based upon a previous design, the Therac-25 was an *improved* therapy system that could deliver two different kinds of radiation: either a low-power electron beam (beta particles) or X-rays. Because of a subtle bug called a *race condition*, a quick-fingered typist could accidentally configure the Therac-25 so the electron beam would fire in high-power mode but with the metal X-ray target out of position.

January 15, 1990 – AT&T Network Outage: A bug in a new release of the software that controls AT&T's long distance switches causes these mammoth computers to crash when they receive a specific message from one of their neighboring machines. One day a switch in New York crashes and reboots, causing its neighboring switches to crash, then their neighbors' neighbors, and so on. Soon, 114 switches are crashing and rebooting every six seconds, leaving an estimated 60 thousand people without long distance service for nine hours. The fix: engineers load the previous software release.

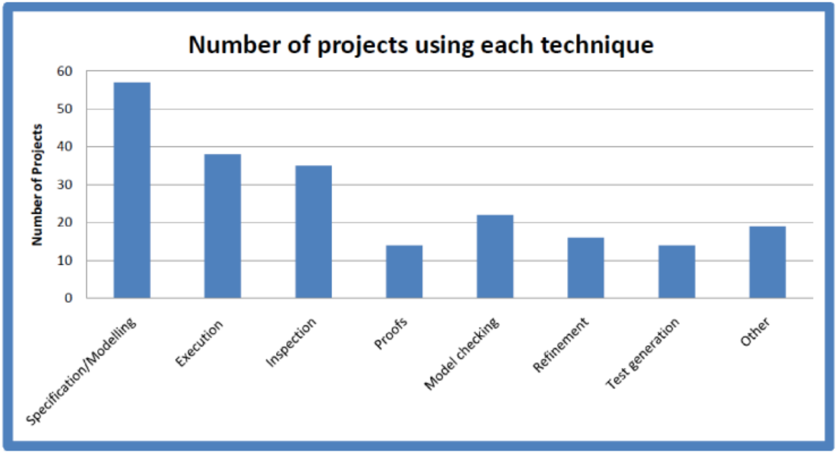
1993 – Intel Pentium floating point divide: A silicon error causes Intel's highly promoted Pentium chip to make mistakes when dividing floating-point numbers that occur within a specific range. At first Intel only offers to replace Pentium chips for consumers who can prove that they need high accuracy; eventually the company relents and agrees to replace the chips for anyone who complains. The bug ultimately costs Intel $475 million.

1995/1996 – The Ping of Death: A lack of sanity checks and error handling in the IP fragmentation reassembly code makes it possible to crash a wide variety of operating systems by sending a malformed *ping* packet from anywhere on the internet. Most obviously affected are computers running Windows, which lock up and display the so-called *blue screen of death* when they receive these packets. But the attack also affects many Macintosh and Unix systems as well.
November 2000 – National Cancer Institute, Panama City: In a series of accidents, therapy planning software created by Multidata Systems International, a U.S. firm, miscalculates the proper dosage of radiation for patients undergoing radiation therapy. At least eight patients die, while another 20 receive overdoses likely to cause significant health problems. The physicians, who were legally required to double-check the computer's calculations by hand, are indicted for murder.

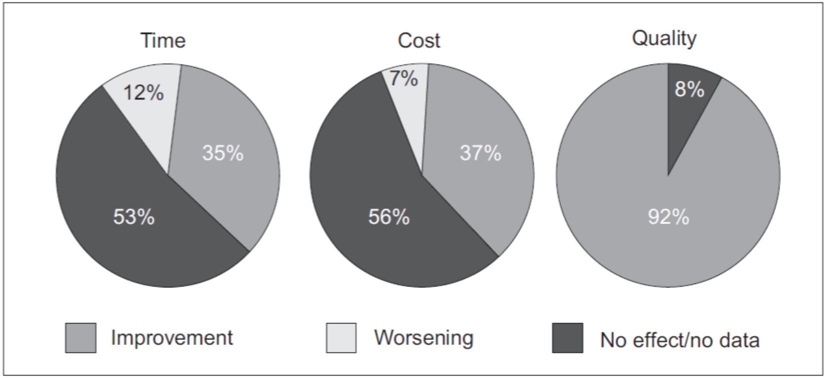
Business and Formal Methods



Techniques


Number of projects using each technique
Number of Projects
60
50
40
30
20
10
0
Specification/Modelling
Execution
Inspection
Proofs
Model checking
Refinement
Test generation
Other

Outcomes


Time
12%
35%
53%
Cost
7%
37%
56%
Quality
8%
92%
Improvement
Worsening
No effect/no data

Famous Projects

Railway Signaling: The B Formal Method has been used for Line 14 of the Paris Metro, a system in use since October 1998 and for the driverless Paris Roissy Airport shuttle, in use since 2007.

Airbus: Airbus have used SCADE for the last ten years for the development of DO-178B Level A controllers for the A340-500/600 series, including the Flight Control Secondary Computer and the Electric Load Management Unit.

The Maeslant Kering Storm Surge Barrier: The Maeslant Kering is a movable barrier protecting the port of Rotterdam from flooding as a result of adverse weather and sea conditions. Data and operations were modelled in Z and this was embedded into a Promela model describing control, and designs were validated using the SPIN model checker

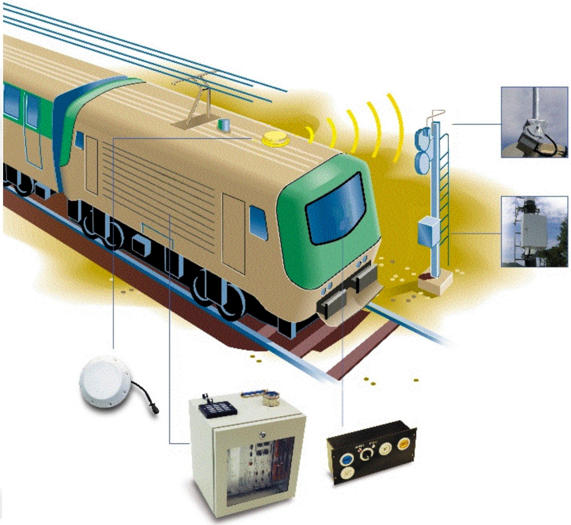
Your Today Challenge

I understand formal methods are cool and I want to use them for the development of my system, BUT...

- ...when shall I use formal methods?
- ...which formal method shall I choose?
- ...what and how shall be formally modeled?
- ...how shall I perform formal verification?
- ...which tool shall I choose?

A case study will help answering these questions...

Case Study: Train Stop System



System Requirement(s)

Functional Requirement

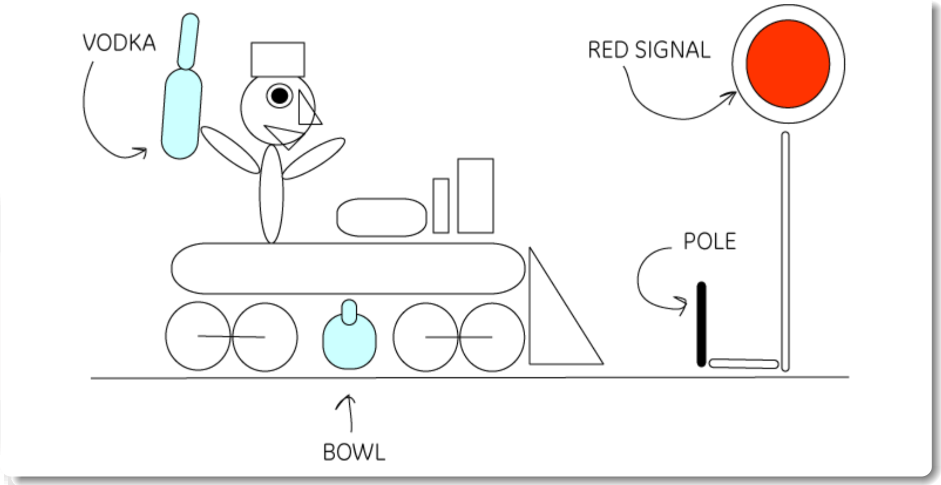
The system shall brake the train when a red signal is passed

Safety Requirement

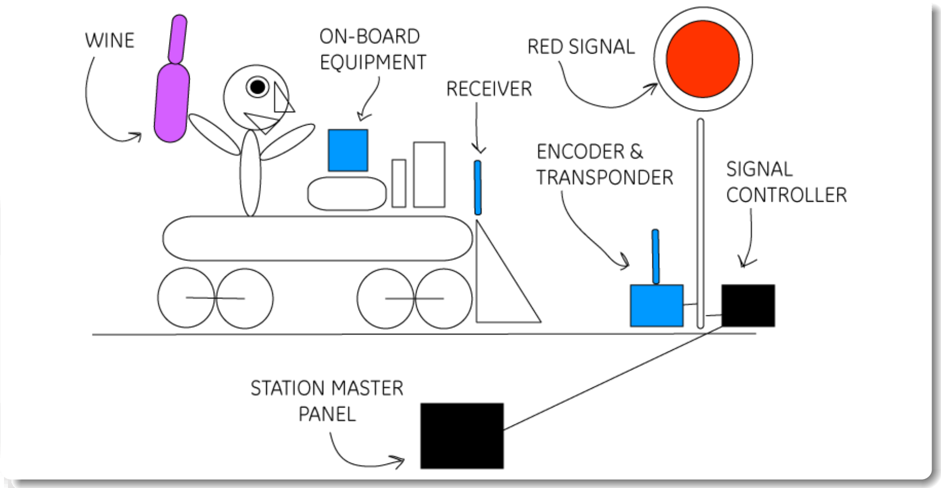
In case a failure occurs in the system, the train and the other devices shall go to their safe state

Which are the safe states?
What if I had an aircraft?

...what about the Russian way?



...Italians (try to) do it better!



System Architecture

- Station Master panel: the Station Master sends a *red signal* command to the signal controller
- Signal and signal controller: the signal controller changes the signal aspect
- Encoder and Transponder: the encoder translates the signal aspect into a message for the on-board receiver and sends the message through the transponder
- On-board receiver: receives the message and sends it to the on-board equipment
- On-board equipment: processes the signal and takes decisions accordingly

Where will we apply formal methods?

Where will we apply formal methods?

On-board software

The on-board software implements the algorithms needed to brake the train in case of SPAD

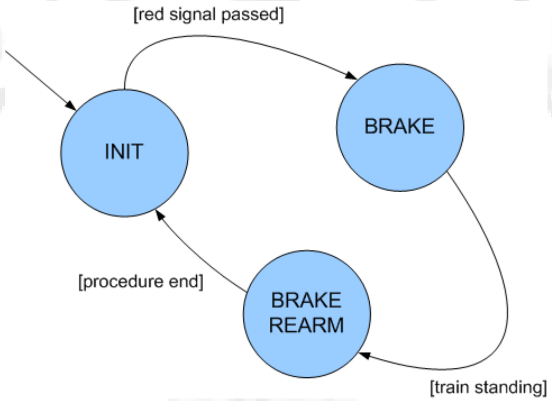
On-board and wayside hardware

We have chosen to use the same hardware for both the on-board equipment and the encoder

Communication protocols

- between panel and signal controller
- between transponder and train

On-board software: the Braking Procedure



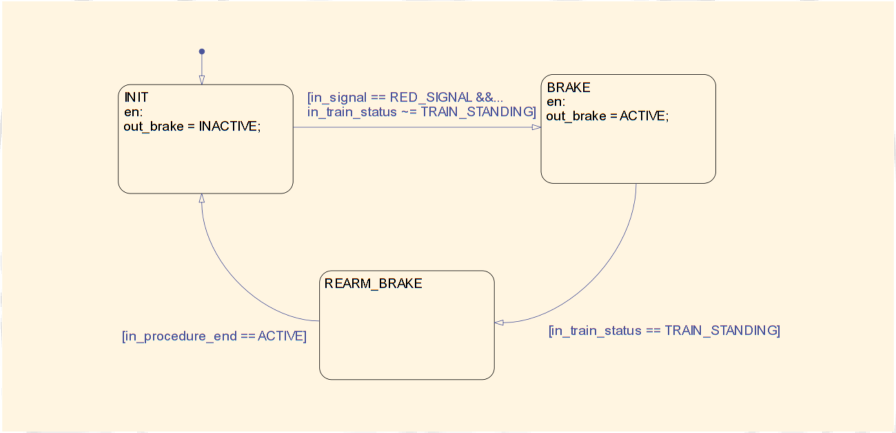
Functional Requirements

1. The system shall issue a brake command when a red signal is passed and if the train is not standing
2. If no red signal is received the braking command shall not be activated
3. The system shall allow brake rearm only when the train is standing
4. When the brake has been rearmed the braking command can be de-activated

...we will use Simulink Design Verifier

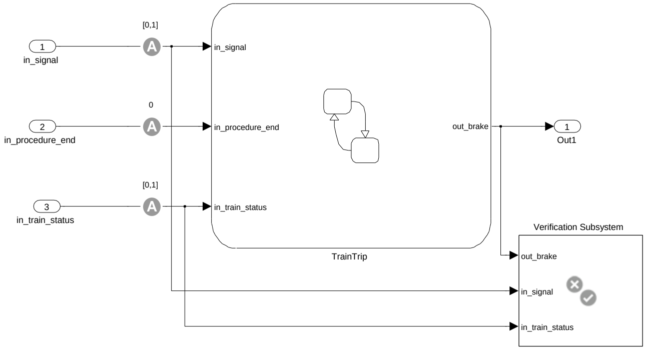
- Platform for Property Proving, Bounded Model Checking, Test Generation
- Verification only for sequential models
- Visual language for modeling (Simulink/Stateflow)
- Well suited for modeling and verifying single-task applications
- Verification of assertions expressed with the same modeling language used for the models
- Proprietary tool
- Proprietary algorithm (Prover)
- Needs Matlab to run
- `http://www.mathworks.com`

Model Definition


INIT
en:
out_brake = INACTIVE;
[in_signal == RED_SIGNAL &&...
in_train_status ~= TRAIN_STANDING]
BRAKE
en:
out_brake = ACTIVE;
[in_train_status == TRAIN_STANDING]
REARM_BRAKE
[in_procedure_end == ACTIVE]

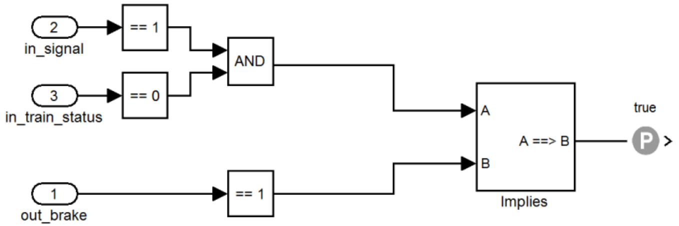
Model Verification

...verification can be performed only on intput/output variables



Model Verification

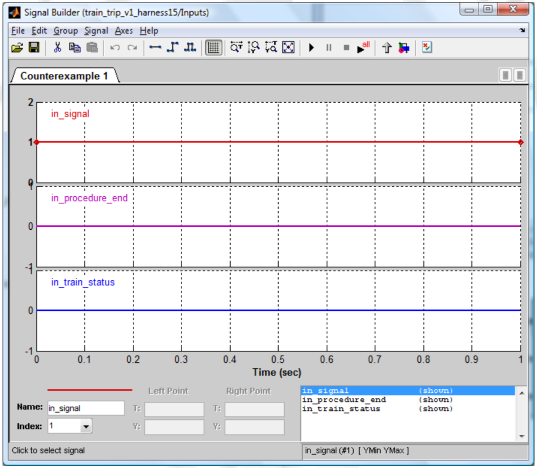
The system shall issue a brake command when a red signal is passed and if the train is not standing



...is it correct?

Model Verification

...no? why?



Name: in_signal
Index: 1

```
in_signal             (shown)
in_procedure_end      (shown)
in_train_status       (shown)
```

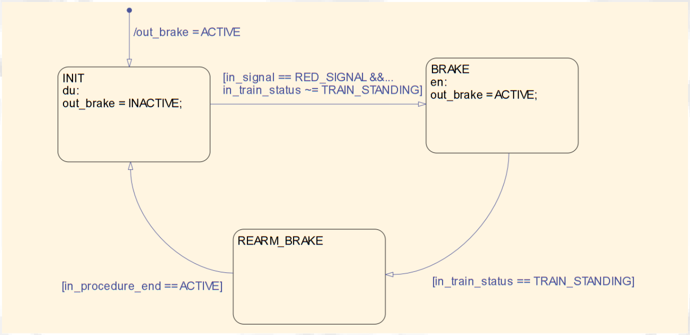
Click to select signal

in_signal (#1) [YMin YMax]

Model Refinement

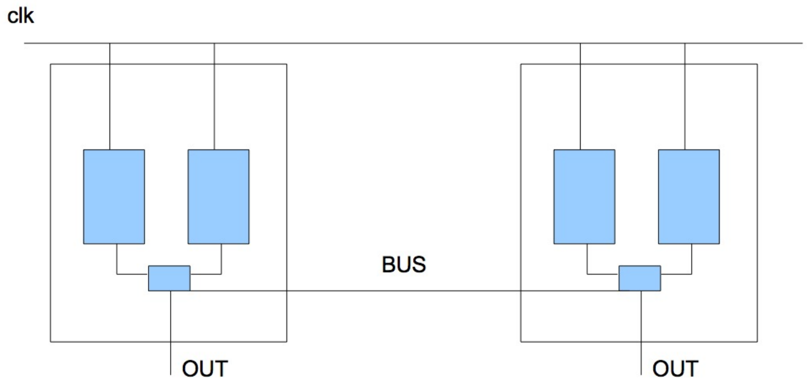
Additional Requirement

During system startup the brake shall be active


/out_brake = ACTIVE
INIT
du:
out_brake = INACTIVE;
[in_signal == RED_SIGNAL &&...
in_train_status ~= TRAIN_STANDING]
BRAKE
en:
out_brake = ACTIVE;
REARM_BRAKE
[in_procedure_end == ACTIVE]
[in_train_status == TRAIN_STANDING]

...what if I would set brake to active when entering the INIT state?

On-board and Wayside Hardware: Hot Stand-by



Non-Functional Requirements

1. The system shall be composed of two syncronous board
2. Each board shall be structured with a synchronous two out of two architecture
3. Each processor executes the same program
4. A board which is capable of driving outputs is called Master
5. A board not capable of driving outputs is called Slave
6. During startup the Master board shall be randomly chosen

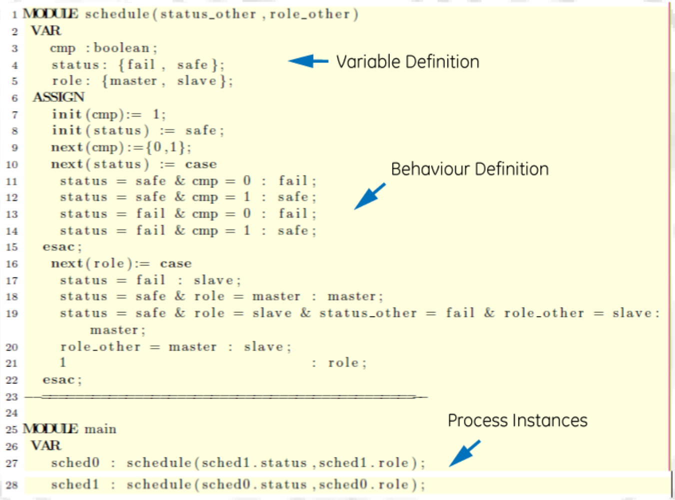
Functional Requirements

1. At each execution step only one board shall be in Master mode
2. If the comparator of one board reveals a failure, it shall signal the failure to the other board if this is active and if it is not signalling a failure itself
3. If the comparator of one board reveals a failure, it shall go into Failure mode
4. If the Slave board receives a failure message from the Master board, and its comparator is not revealing any failure, it shall switch to Master mode

...we will use NuSMV

- Textual modeling language for finite state transition systems (synchronous and asynchronous)
- Well suited for modeling hardware circuits
- Symbolic model checking, Bounded Model Checking
- Based on Binary Decision Diagrams (BDD) e Propositional Satisfiability (SAT) solvers
- Verification of LTL and CTL properties
- Open source!
- No additional tool needed
- `http://nusmv.irst.itc.it`

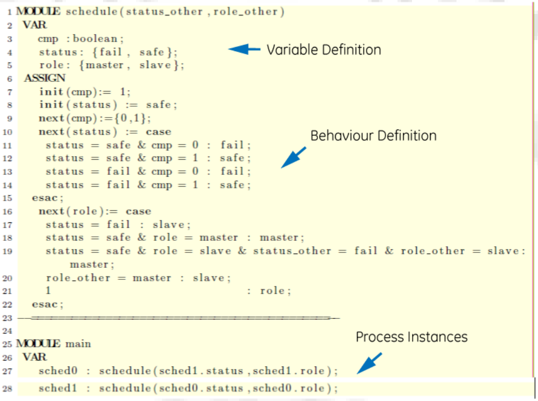
Model Definition



```
MODULE schedule(status_other,role_other)
 VAR
   cmp :boolean;
   status: {fail, safe};
   role: {master, slave};
 ASSIGN
   init(cmp):= 1;
   init(status) := safe;
   next(cmp):={0,1};
   next(status) := case
    status = safe & cmp = 0 : fail;
    status = safe & cmp = 1 : safe;
    status = fail & cmp = 0 : fail;
    status = fail & cmp = 1 : safe;
  esac;
   next(role):= case
    status = fail  : slave;
    status = safe & role = master : master;
    status = safe & role = slave & status_other = fail & role_other = slave:
       master;
    role_other = master : slave;
    1                               : role;
  esac;
----------------------------------------------

MODULE main
 VAR
   sched0 : schedule(sched1.status,sched1.role);
   sched1 : schedule(sched0.status,sched0.role);
```

...why didn’t we model the processors?

Model Verification



```
MODULE schedule(status_other,role_other)
 VAR
   cmp :boolean;
   status: {fail, safe};
   role: {master, slave};
 ASSIGN
   init(cmp):= 1;
   init(status) := safe;
   next(cmp):={0,1};
   next(status) := case
    status = safe & cmp = 0 : fail;
    status = safe & cmp = 1 : safe;
    status = fail & cmp = 0 : fail;
    status = fail & cmp = 1 : safe;
  esac;
   next(role):= case
    status = fail : slave;
    status = safe & role = master : master;
    status = safe & role = slave & status_other = fail & role_other = slave:
        master;
    role_other = master : slave;
    1                               : role;
  esac;
--==========================================--

MODULE main
 VAR
   sched0 : schedule(sched1.status,sched1.role);
   sched1 : schedule(sched0.status,sched0.role);
```

At each execution step only one board shall be in Master mode

```
CTLSPEC AG ! (sched0.role=master & sched1.role=master)
```

Model Verification

At each execution step only one board shall be in Master mode

```
CTLSPEC AG ! (sched0.role=master & sched1.role=master)
```

```
-- specification AG !(sched0.role = master & sched1.role = master)  is false
-- as demonstrated by the following execution sequence
Trace Description: CTL Counterexample
Trace Type: Counterexample
-> State: 1.1 <-
  sched0.cmp = 1
  sched0.status = safe
  sched0.role = master
  sched1.cmp = 1
  sched1.status = safe
  sched1.role = master
```

...how do we fix this?

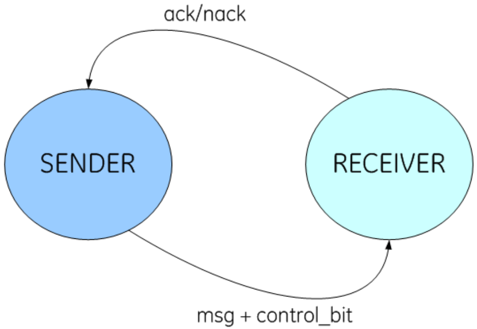
Model Refinement

We forgot to implement one of the requirements...

```
MODULE main
 VAR
   startAsMaster:boolean;
   sched0  :  schedule(sched1.status,startAsMaster);
   sched1  : schedule(sched0.status,!startAsMaster);

--==========================================================--
```

What if this happened later in the development?
Is this solution applicable?

Panel to Signal Protocol: Alternating Bit Protocol



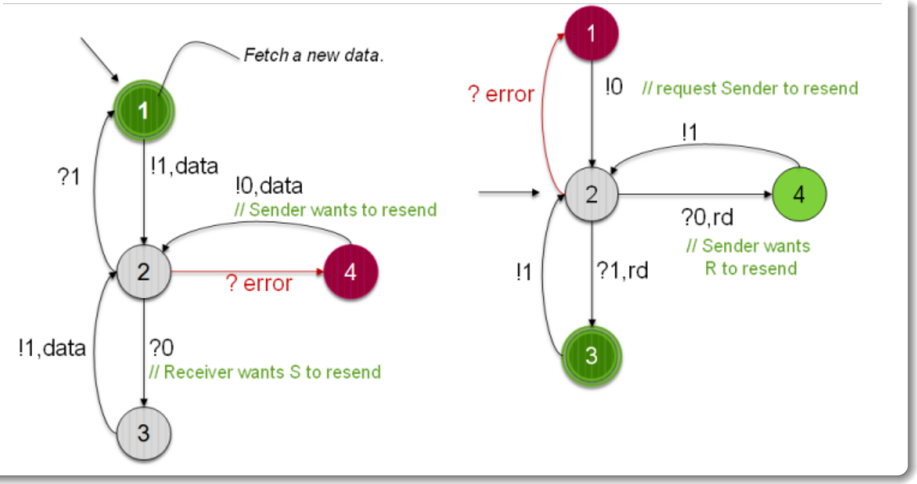
Requirements

1. The Sender shall continuously send the signal status to the signal controller
2. The Sender shall send messages to the Receiver together with a control bit
3. When receiving a correct message, the receiver shall answer with an ack bit
4. When receiving an incorrect message, the receiver shall answer with a nack bit
5. If the Sender receives an ack bit, it shall send the next message
6. If the Sender receives a nack bit, it shall re-send the previous message
7. If the Sender receives a wrong message, it shall ask the Receiver to re-send its message
8. The Sender sends the message until the Receiver receives it without errors
9. Each accepted data will be accepted only once

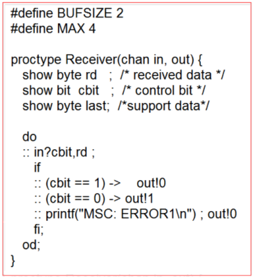
...we will use SPIN

- PROMELA (PROgramming MEta-LAnguage) modeling language
- Well suited for modeling distributed software system and communication protocols
- Explicit model checking
- On the fly model checking
- Verification of LTL properties
- Open source!
- Needs gcc or cygwin with gcc
- XSPIN needs tcl/tk (downloadable through cygwin)
- `http://spinroot.com/spin/whatispin.html`

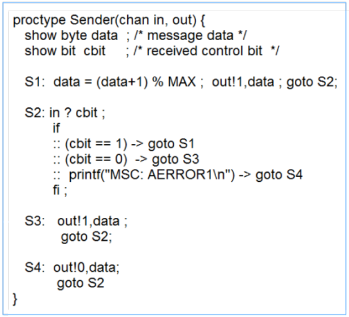
Model Definition


Fetch a new data.
1
!1,data
?1
!0,data
// Sender wants to resend
2
? error
4
!1,data
?0
// Receiver wants S to resend
3
1
? error
!0
// request Sender to resend
!1
2
4
?0,rd
// Sender wants R to resend
!1
?1,rd
3

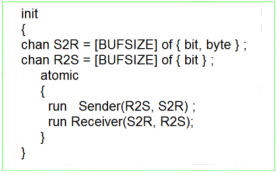
Model Definition



```
#define BUFSIZE 2
#define MAX 4

proctype Receiver(chan in, out) {
  show byte rd   ;  /* received data */
  show bit  cbit  ;  /* control bit */
  show byte last;  /*support data*/

  do
  :: in?cbit,rd ;
    if
    :: (cbit == 1) ->    out!0
    :: (cbit == 0) -> out!1
    :: printf("MSC: ERROR1\n") ; out!0
    fi;
  od;
}
```



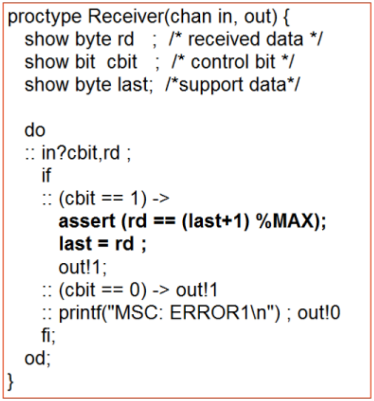
```
proctype Sender(chan in, out) {
  show byte data  ; /* message data */
  show bit  cbit     ; /* received control bit  */

  S1:  data = (data+1) % MAX ;  out!1,data ; goto S2;

  S2: in ? cbit ;
         if
         :: (cbit == 1) -> goto S1
         :: (cbit == 0)  -> goto S3
         ::  printf("MSC: AERROR1\n") -> goto S4
         fi ;

  S3:   out!1,data ;
          goto S2;

  S4:  out!0,data;
         goto S2
}
```



```
init
{
chan S2R = [BUFSIZE] of { bit, byte } ;
chan R2S = [BUFSIZE] of { bit } ;
     atomic
     {
       run   Sender(R2S, S2R) ;
       run Receiver(S2R, R2S);
     }
}
```

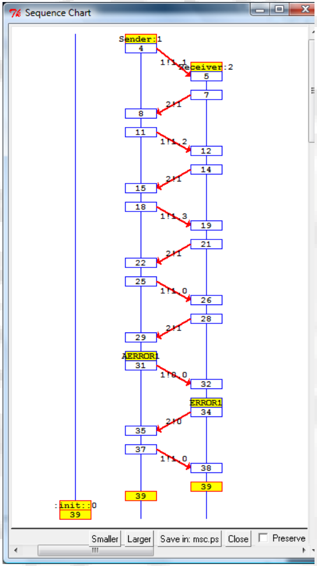
Model Verification

Each accepted data will be accepted only once



```
proctype Receiver(chan in, out) {
  show byte rd   ;  /* received data */
  show bit  cbit  ;  /* control bit */
  show byte last;  /*support data*/

  do
  :: in?cbit,rd ;
    if
    :: (cbit == 1) ->
      assert (rd == (last+1) %MAX);
      last = rd ;
      out!1;
    :: (cbit == 0) -> out!1
    :: printf("MSC: ERROR1\n") ; out!0
    fi;
  od;
}
```

Counterexample

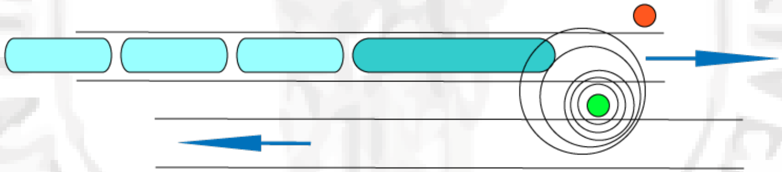


- The Receiver sends the ACK
- The Sender looses the message and ask another ACK
- The Receiver looses the message and ask to resend
- The Sender sends again the same message
- ...was the requirement actually needed?

Transponder to Train Protocol

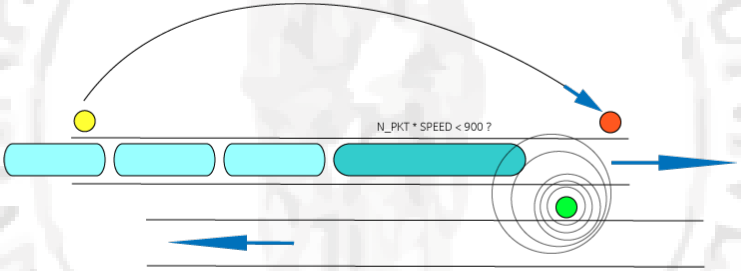
What's needed?

- The communication protocol shall be fast (no ack/nack)
- We have to consider that a message can be missed
- We have no consider that the train can receive wrong messages
- We have to consider that the train can receive messages coming from the opposite direction



Architectural Solution

- Unidirectional communication
- Appointment and missed appointment concepts
- Airgap concept (N_PKT * SPEED)



Some answers to our previous questions

- ...when shall I use formal methods? As soon as possible!
- ...which formal method shall I choose? Balance efficiency and safety of the method according to the application
- ...what and how shall be formally modeled? Only what needed and with the right degree of granularity
- ...how shall I perform formal verification? Considering that errors could be in requirements, model and formulas
- ...which tool shall I choose? No right answer...Design Verifier for sequential SW, NuSMV for HW and SPIN for communication is a reasonable hypothesis

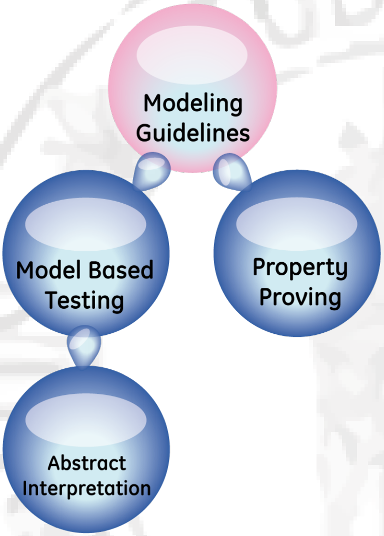
...and Another Question

...What shall we do with the wayside software?

...and Another Question

...What shall we do with the wayside software?

- The wayside software shall take care of setting the safe state of the signal in case of communication failures with the panel
- The safe state for the signal is red
- The signal shall go into its safe state when de-energized
- We could use Design Verifier



Modeling Guidelines

- Semantics restrictions

Model Based Testing and Abstract Interpretation

- Functional coherence
- Runtime error freedom

Property Proving

- Formal verification