A Systematic Approach to Requirements Driven Test Generation for Safety Critical Systems

Toby Wilkinson Michael Butler John Colley

Electronics and Computer Science
University of Southampton, SO17 1BJ, United Kingdom
stw08r@ecs.soton.ac.uk

IMBSA 2014

Outline

Outline

Model Based Testing

Model Based Testing (MBT) is not a new idea.

However two issues remain:

1. How do you build the model?
    1.1 What do you model?
    1.2 How do you demonstrate that the model is correct?
    1.3 What are the real requirements?
    1.4 Are the requirements consistent?
    1.5 Are there missing requirements?
2. How do you generate an adequate, yet efficient/tractable test set form the model?
    2.1 Controllers typically intended to run "forever".
    2.2 Models therefore contain infinite behaviour.
    2.3 Exhaustive testing not possible.
    2.4 Need to select a subset of tests - but which?

Our Work

Our work looks at both these issues.

The work is ongoing, but we have two projects that are just finishing:

HASTE
: A project funded by Dstl the UK Defence Science and Technology Laboratory.

  Interested in using:

  1. COTS components in safety critical systems.
  2. Reusing proven components in new contexts.

  University of Leicester have developed a complementary technique using machine learning.

ADVANCE
: An FP7 project (`http://www.advance-ict.eu`).

  Focuses on applying formal methods to cyber-physical systems.

Outline

Event-B

Event-B is a formal method developed by Jean-Raymond Abrial.

It aims to facilitate:

1. Working at the System Level.
2. Formalise Requirements Engineering.
3. Model refinement - this has a **precise** mathematical meaning.

Event-B is **PROOF** based. Properties are proved mathematically, **NOT** checked by a model checker.

(Though we do have a model checker, and it is a vital part of the toolset - more on this later.)

Event-B Example

Event-B uses **contexts** to define types:

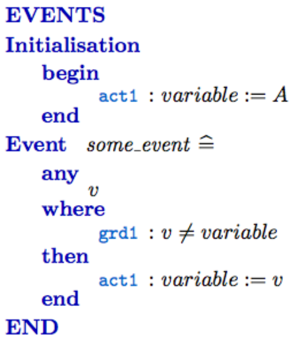
```
CONTEXT abstract_context
SETS
    SOME_TYPE
CONSTANTS
    A
    B
AXIOMS
    axm1 : partition(SOME_TYPE, {A}, {B})
END
```

Models are constructed as Event-B **machines**:

```
MACHINE abstract_machine
SEES abstract_context
VARIABLES
    variable
INVARIANTS
    inv1 : variable ∈ SOME_TYPE
```

Events in Event-B

Event-B models have **Events**:



```
EVENTS
Initialisation
    begin
        act1 : variable := A
    end
Event  some_event ≙
    any
        v
    where
        grd1 : v ≠ variable
    then
        act1 : variable := v
    end
END
```

Events are **guarded atomic actions**.

Safety/Security Properties in Event-B

Safety and security properties are expressed in Event-B as **invariants**.

Aircraft Landing Gear: The landing gear can only extend if the gear doors are open

$$Gear = extending \Rightarrow Door = open$$

Railway Signalling Safety: The signal of a route can only be green when all blocks of that route are unoccupied

$$sig(r) = GREEN \Rightarrow blocks[r] \cap occupied = \emptyset$$

Access Control in Secure Building: If user u is in room r, then u must have sufficient authority to be in r

$$location(u) = r \Rightarrow takeplace[r] \subseteq authorised[u]$$

Proof Obligations

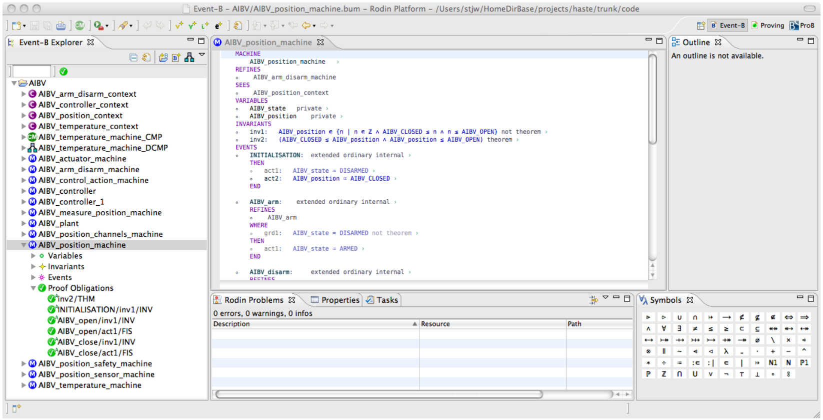
For each Event-B machine **proof obligations** are generated.

Some of the main ones are:

Invariant Preservation
: Every event must preserve the invariants. If an event does not preserve an invariant then the guard must be **strengthened**.

Machine Refinement
: Every event in a refined machine must refine an event in the more abstract machine (or the skip event), and that abstract event must be enabled whenever the refined event is enabled.

Rodin



Event-B – AIBV/AIBV_position_machine.bum – Rodin Platform – /Users/stjw/HomeDirBase/projects/haste/trunk/code

Event-B Explorer

```
AIBV
  AIBV_arm_disarm_context
  AIBV_controller_context
  AIBV_position_context
  AIBV_temperature_context
  AIBV_temperature_machine_CMP
  AIBV_temperature_machine_DCMP
  AIBV_actuator_machine
  AIBV_arm_disarm_machine
  AIBV_control_action_machine
  AIBV_controller
  AIBV_controller_1
  AIBV_measure_position_machine
  AIBV_plant
  AIBV_position_channels_machine
  AIBV_position_machine
    Variables
    Invariants
    Events
    Proof Obligations
      inv2/THM
      INITIALISATION/inv1/INV
      AIBV_open/inv1/INV
      AIBV_open/act1/FIS
      AIBV_close/inv1/INV
      AIBV_close/act1/FIS
  AIBV_position_safety_machine
  AIBV_position_sensor_machine
  AIBV_temperature_machine
```

AIBV_position_machine

```
MACHINE
    AIBV_position_machine ›
REFINES
∘    AIBV_arm_disarm_machine
SEES
∘    AIBV_position_context
VARIABLES
∘   AIBV_state   private ›
∘   AIBV_position    private ›
INVARIANTS
∘   inv1:   AIBV_position ∈ {n | n ∈ ℤ ∧ AIBV_CLOSED ≤ n ∧ n ≤ AIBV_OPEN} not theorem ›
∘   inv2:   (AIBV_CLOSED ≤ AIBV_position ∧ AIBV_position ≤ AIBV_OPEN) theorem ›
EVENTS
∘   INITIALISATION:  extended ordinary internal ›
    THEN
    ∘   act1:   AIBV_state ≔ DISARMED ›
    ∘   act2:   AIBV_position ≔ AIBV_CLOSED ›
    END

∘   AIBV_arm:    extended ordinary internal ›
    REFINES
    ∘    AIBV_arm
    WHERE
    ∘   grd1:   AIBV_state = DISARMED not theorem ›
    THEN
    ∘   act1:   AIBV_state ≔ ARMED ›
    END

∘   AIBV_disarm:     extended ordinary internal ›
```

Outline

An outline is not available.

Rodin Problems | Properties | Tasks

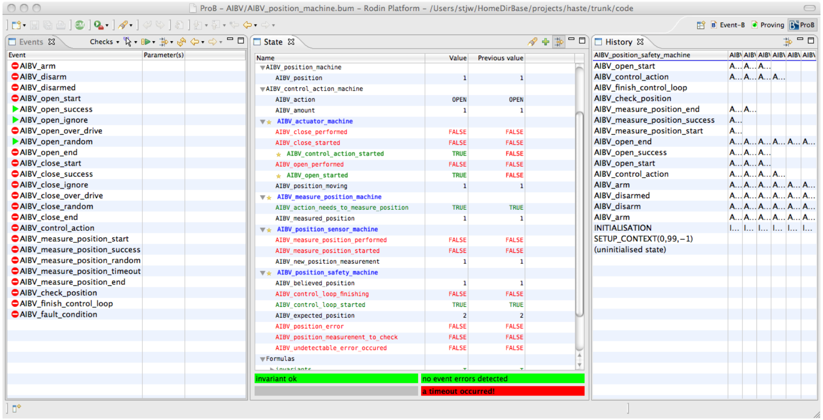
0 errors, 0 warnings, 0 infos

Description	Resource	Path

Eclipse based development environment.

Automatically generates proof obligations and **attempts** to discharge them.

ProB



ProB – AIBV/AIBV_position_machine.bum – Rodin Platform – /Users/stjw/HomeDirBase/projects/haste/trunk/code
Event-B
Proving
ProB
Events
Checks
State
History

Event	Parameter(s)
⊖ AIBV_arm	
⊖ AIBV_disarm	
⊖ AIBV_disarmed	
⊖ AIBV_open_start	
▶ AIBV_open_success	
▶ AIBV_open_ignore	
⊖ AIBV_open_over_drive	
▶ AIBV_open_random	
⊖ AIBV_open_end	
⊖ AIBV_close_start	
⊖ AIBV_close_success	
⊖ AIBV_close_ignore	
⊖ AIBV_close_over_drive	
⊖ AIBV_close_random	
⊖ AIBV_close_end	
⊖ AIBV_control_action	
⊖ AIBV_measure_position_start	
⊖ AIBV_measure_position_success	
⊖ AIBV_measure_position_random	
⊖ AIBV_measure_position_timeout	
⊖ AIBV_measure_position_end	
⊖ AIBV_check_position	
⊖ AIBV_finish_control_loop	
⊖ AIBV_fault_condition	

Name	Value	Previous value
▼AIBV_position_machine		
AIBV_position	1	1
▼AIBV_control_action_machine		
AIBV_action	OPEN	OPEN
AIBV_amount	1	1
▼★ AIBV_actuator_machine		
AIBV_close_performed	FALSE	FALSE
AIBV_close_started	FALSE	FALSE
★ AIBV_control_action_started	TRUE	FALSE
AIBV_open_performed	FALSE	FALSE
★ AIBV_open_started	TRUE	FALSE
AIBV_position_moving	1	1
▼★ AIBV_measure_position_machine		
AIBV_action_needs_to_measure_position	TRUE	TRUE
AIBV_measured_position	1	1
▼★ AIBV_position_sensor_machine		
AIBV_measure_position_performed	FALSE	FALSE
AIBV_measure_position_started	FALSE	FALSE
AIBV_new_position_measurement	1	1
▼★ AIBV_position_safety_machine		
AIBV_believed_position	1	1
AIBV_control_loop_finishing	FALSE	FALSE
AIBV_control_loop_started	TRUE	TRUE
AIBV_expected_position	2	2
AIBV_position_error	FALSE	FALSE
AIBV_position_measurement_to_check	FALSE	FALSE
AIBV_undetectable_error_occured	FALSE	FALSE
▼Formulas		

invariant ok

no event errors detected

a timeout occurred!

AIBV_position_safety_machine	AIB\	AIB\	AIB\	AIB\	AIB\	AIB\
AIBV_open_start	A...	A...	A...			
AIBV_control_action	A...	A...	A...	A...		
AIBV_finish_control_loop						
AIBV_check_position						
AIBV_measure_position_end	A...	A...				
AIBV_measure_position_success	A...					
AIBV_measure_position_start	A...					
AIBV_open_end	A...	A...	A...	A...	A...	A...
AIBV_open_success	A...	A...	A...			
AIBV_open_start	A...	A...	A...			
AIBV_control_action	A...	A...	A...	A...		
AIBV_arm	A...	A...	A...	A...	A...	A...
AIBV_disarmed	A...	A...	A...	A...	A...	A...
AIBV_disarm	A...	A...	A...	A...	A...	A...
AIBV_arm	A...	A...	A...	A...	A...	A...
INITIALISATION	I...	I...	I...	I...	I...	I...
SETUP_CONTEXT(0,99,−1)						
(uninitialised state)						

Rodin incorporates the ProB model checker which can be used to:

1. Find counter examples to proof obligations.
2. **Animate** the model to help with model **validation**.

Safety Requirements

Safety requirements in Event-B are:

1. Modelled as **invariants**.
2. Preserved by adding event **guards**.

But guards must not eliminate events that **CAN** occur, in particular, failures in the **environment**: sensor failures, actuator failures, lost messages etc.

If this means the safety invariants are not preserved then either:

1. Hazard **mitigation** or **elimination** strategies must be incorporated into the model.
2. The safety requirement must be modified to reflect what can **actually** be acheived – i.e. the system is safe **assuming** a particular sequence of failures does not occur.

   This makes the assumption **explicit**.

Finding Safety Requirements

How do we find safety requirements?

We use a process called STPA:

1. Developed at MIT by Nancy Leveson.
2. Takes a system-theoretic view.
3. Focuses on the interactions between entities.
4. Fits well with the Event-B method.

Any method though can be used, so long as it finds **all** the safety constraints, and identifies **failures** that can cause **hazards**.

Probably requires input from **domain experts**.

System-Theoretic Process Analysis (STPA)

STPA is a hazard analysis technique based upon Systems Theory and utilises 3 concepts:

1. Safety constraints.
2. Hierarchical control structures.
3. Process models with feedback.

STPA has two steps:

1. Identify potentially **hazardous** control actions and derive the safety constraints.
2. Determine **how** unsafe control actions could occur.

Engineering a Safer World, Leveson, 2011

Outline

Safety Arguments

When generating tests from the model...

I don't want the tool to have **discretion** over which tests are generated.

Because...

I don't want to have to **explain** to a certifier what the tool has done.

I don't want to have to **justify** the choices the tool has made.

But I do **expect** to have to justify the decisions I have made!

Test Generation in Rodin (at the Moment)

Rodin does have the MBT (Model Based Testing) plug-in developed on the DEPLOY project.

The MBT plug-in has limitations for safety critical systems:

1. Approximates state space with a finite state machine.
2. Cannot (automatically) trace approximation to requirements (DO-178B/C) - which states has it missed out?

What can we do to fix this?

Standard Testing Techniques

Exploring **full** set of behaviours **impossible**:

1. Infinite paths in controllers
   – intended to run forever.
2. Large range of input values
   – integers, reals.
3. Possibly unbounded size of input data structures
   – arbitrarily long lists.

Need to **restrict** to a **finite** set of behaviours to be tested:

1. Bound number of control loops performed.
2. Partition input values into equivalence classes
   – test only one value from each class.
3. Bound size of input data structures tested.

Tested behaviour must **cover** the requirements (DO-178B/C)
– especially hazard elimination and mitigation.

Ghost Variables

Introduce ghost variables to the model to facilitate test generation:

Control Loops: Add counter to recorded how many control loops have been performed.

Equivalence Classes: For each system input define the set of equivalence classes to be tested.

Add a variable that records the equivalence class of the actual input value.

Hazardous States: Add a variable that records if a hazardous state has been reached.

Test Generation (a Work in Progress)

Tests are generated using the ProB model checker.

Use the ghost variables to constrain the behaviour of the model in line with your testing plan:

1. Bounded number of control loops.
2. Input equivalence partitioning.

You must **justify** these choices.

ProB attempts to find **all** the paths from the initial state within the constrained model.

Each path contains a sequence of input / output events
– i.e. a sequence of **tests**.

Environment Coverage

To ensure safety, at a minimum, **all** environment events must be **covered**:

1. Different actuator failure modes.
2. Different sensor failure modes.

Events representing failures in the environment were determined during the STPA hazard analysis.

If during modelling it was determined that certain sequences of failures cause hazards, those **sequences** should also be **covered**.

Tests are generated for **all** sequences that satisfy the **constraints** – constraints must **cover** the **environment** events.

You must **justify** the constraints.

Case Study (1/2)

To help develop the techniques described we have conducted several case studies.

On the HASTE project we modelled a controller for an Anti-Ice Bleed Valve for the FADEC (Full Authority Digital Engine Control) system for a hypothetical helicopter.

The case study:

1. Performed the STPA hazard analysis.
2. Constructed an Event-B model of the controller and the valve assembly.
3. The model incorporated events representing sensor and actuator failures identified during the STPA analysis.
4. Tests were generated from the model using ProB.

Case Study (2/2)

For the test generation we:

1. Partitioned the input variable (the ambient air temperature) into 6 equivalence classes (to plot the controller's control law).
2. Bounded the number of control loops to 3 (sufficient to cover all failure sequences).

ProB generated 12,648 test cases in about 15 minutes.

Where a test case is sequence of interactions between the controller and the valve assembly.

The test cases are **short** but cover **all** the corner cases
– good for probing the **boundaries** of the controller's behaviour.

Outline

The Future

Develop a Rodin plug-in tool to provide a testing interface.

Need to determine what such an interface should be like.

Would like to **automate** as much as possible:

1. Adding ghost variables.
2. Constraint generation.

Probably need to prototype first and experiment:

1. Case studies.
2. Industrial collaborations.

Questions

Any Questions?