# PROOF AND TEST WITH RICH SPARK 2014 CONTRACTS

Thomas Wilson, Altran UK

**3 June 2019**

# AGENDA

1 The approach used

2 The system developed

3 Use of contracts during development

4 Use of contracts during static verification

5 Use of contracts during testing

6 Conclusions

# THE APPROACH USED

- This project was our first use of SPARK 2014

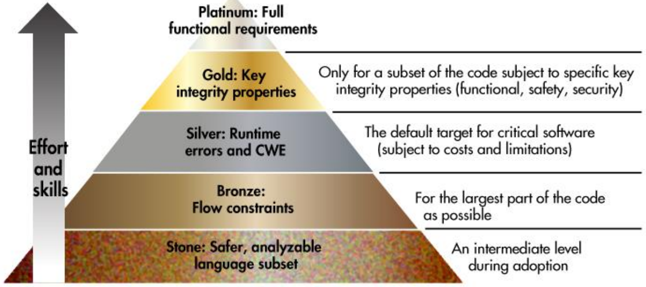
- Previous use of SPARK 2005 and earlier
  - Usually proof of absence of run-time exceptions
  - Contracts provided to support that

- Planned approach for project utilising new capabilities in SPARK 2014
  - Combination of light and heavyweight contracts
  - Combination of proof and test

SPARK 2014

Expanding the boundaries of safe and secure programming.

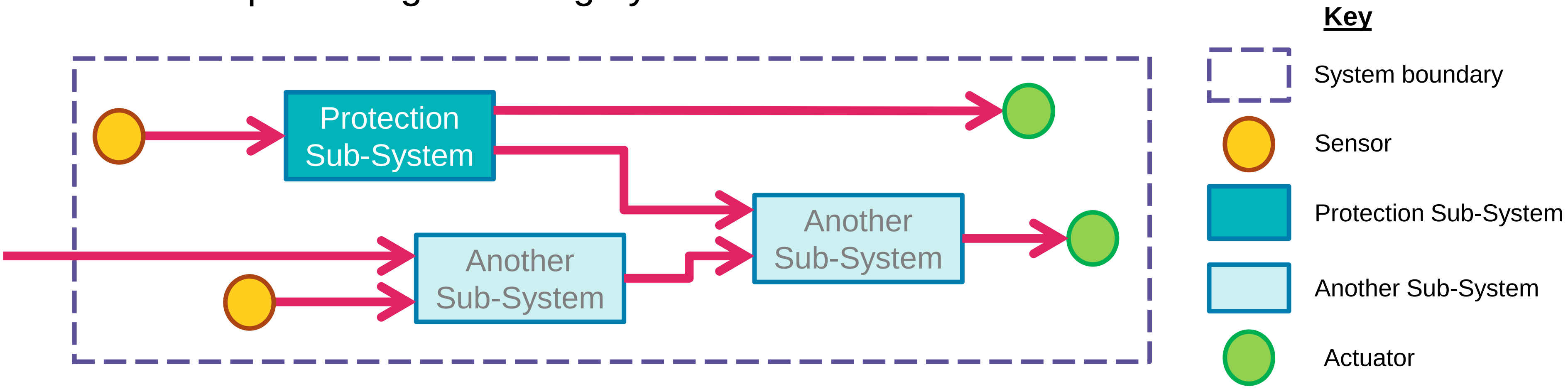
**SPARK Assurance Levels**




*Implementation Guidance for the Adoption of SPARK, AdaCore and Thales*
[https://www.adacore.com/books/implementation-guidance-spark](https://www.adacore.com/books/implementation-guidance-spark)

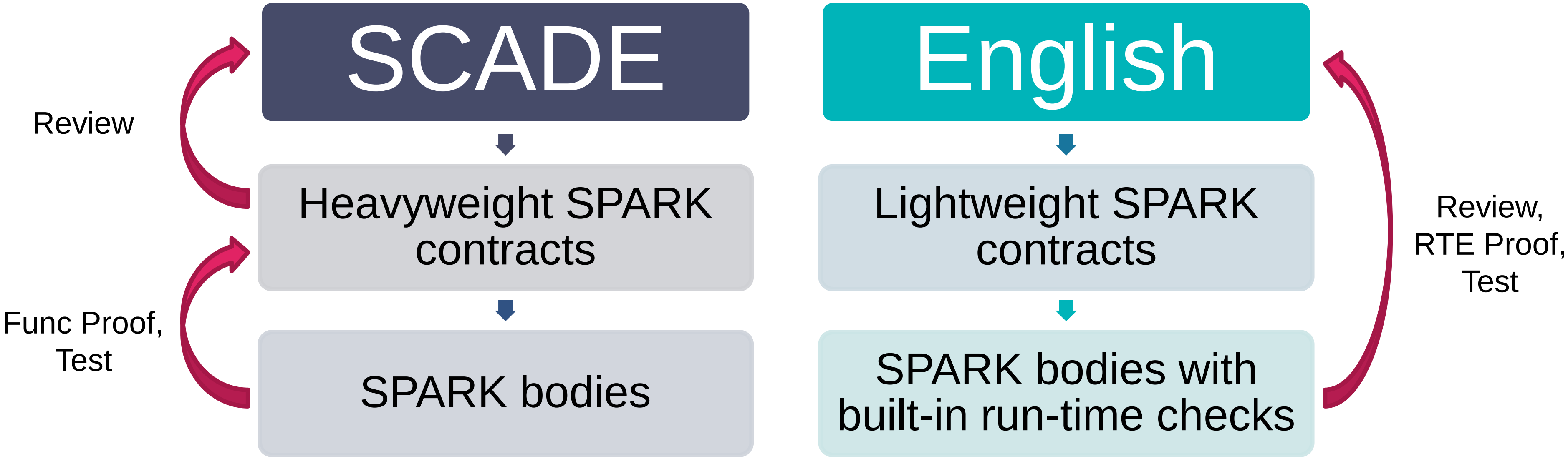
# THE SYSTEM DEVELOPED

- Embedded protection sub-system
  - Monitors operation of a wider system and overrides behaviour if required to maintain safety
  - Developed to highest integrity under UK DEF STAN 00-56

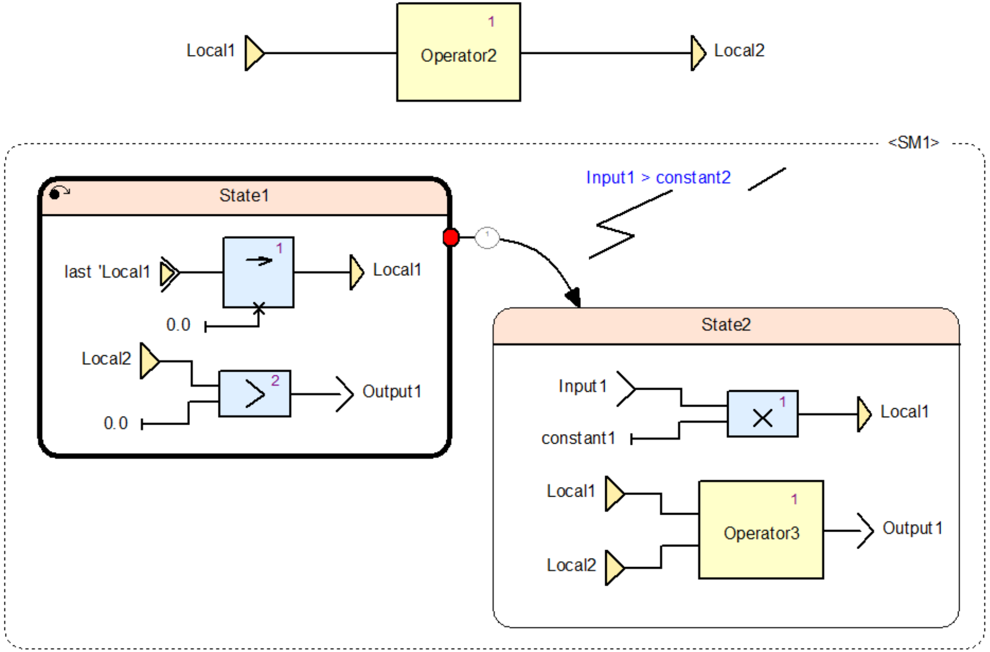



*Model of this type of embedded protection sub-system in context of wider system*

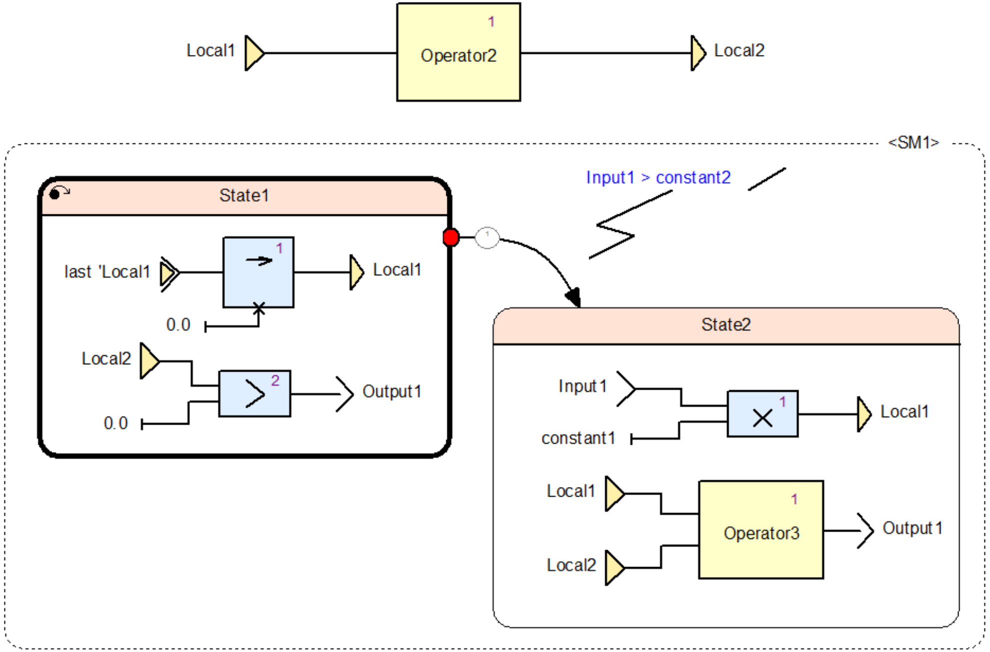
# USE OF CONTRACTS DURING DEVELOPMENT – OVERVIEW



# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS



# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS




Package specification:

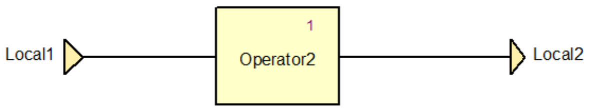
```
package Operator_1
is
   type SM_1_T is (State_1, State_2);

   type State_T is
      record
         Local_1              : Base_Types.Float64;
         Local_2              : Base_Types.Float64;
         Operator_2_1_State   : Operator_2.State_T;
         SM_1                 : SM_1_T;
         Init_1_Evaluated     : Boolean;
         Operator_3_1_State   : Operator_3.State_T;
      end record;

   type Result_T is
      record
         State    : State_T;
         Output_1 : Boolean;
      end record;

   function Initialise return State_T
      with Post => (...);

   function Update (Old_State : State_T;
                    Input_1   : Base_Types.Float64)
      return Result_T
      with Post => (...);

end Operator_1;
```

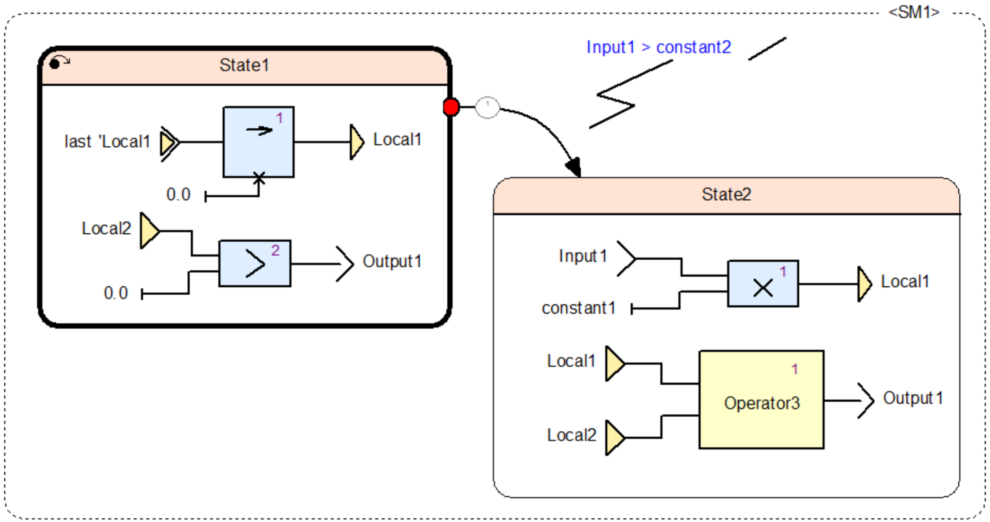
altran

# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS

Update function postcondition:

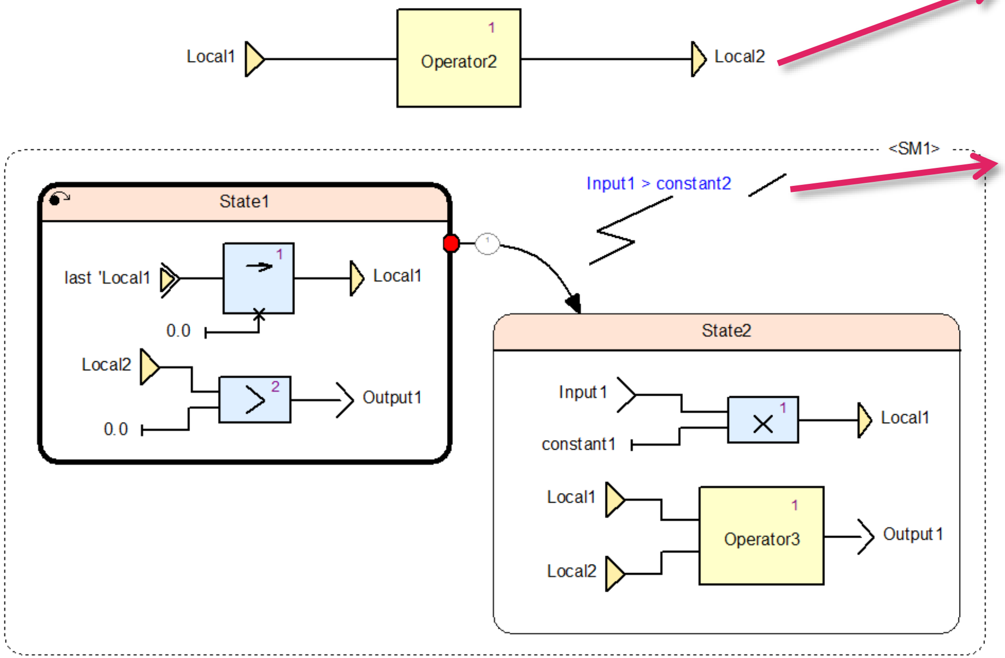
```
(Operator_2.Result_T' (
   State    => Update'Result.State.Operator_2_1_State,
   Output_1 => Update'Result.State.Local_2)
 = Operator_2.Update (
       Old_State => Old_State.Operator_2_1_State,
       Input_1   => Update'Result.State.Local_1)) and
```



# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS

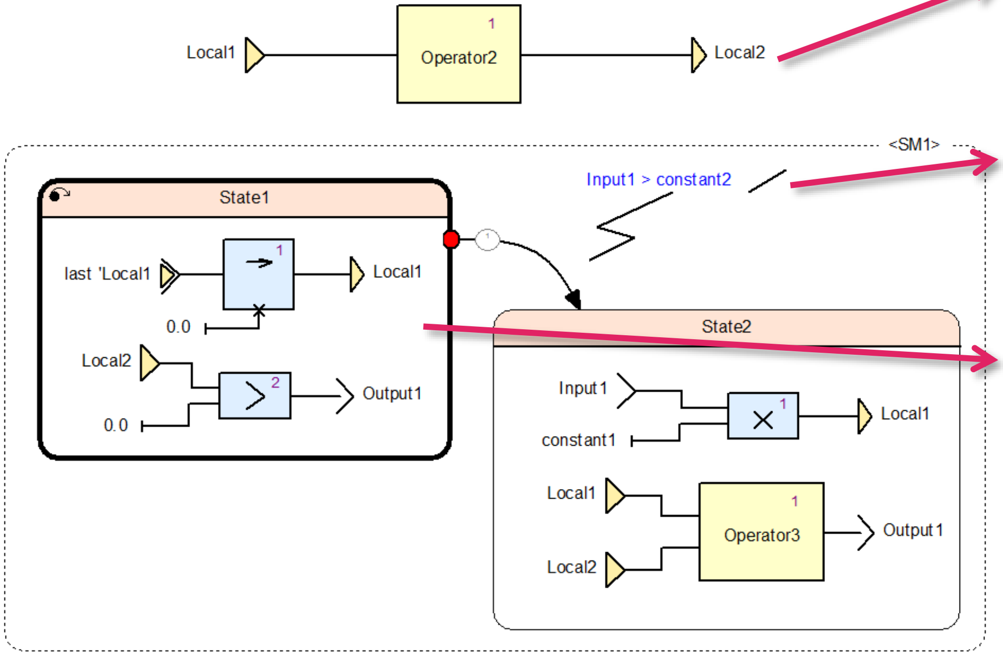
Update function postcondition:




```
(Operator_2.Result_T' (
   State    => Update'Result.State.Operator_2_1_State,
   Output_1 => Update'Result.State.Local_2)
 = Operator_2.Update (
      Old_State => Old_State.Operator_2_1_State,
      Input_1   => Update'Result.State.Local_1)) and
(if Old_State.SM_1 = State_1 then
   (if (Input_1 > Constants.Constant_2) then
      Update'Result.State.SM_1 = State_2
    else
      Update'Result.State.SM_1 = Old_State.SM_1)) and
(if Old_State.SM_1 = State_2 then
   Update'Result.State.SM_1 = Old_State.SM_1) and
```

# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS

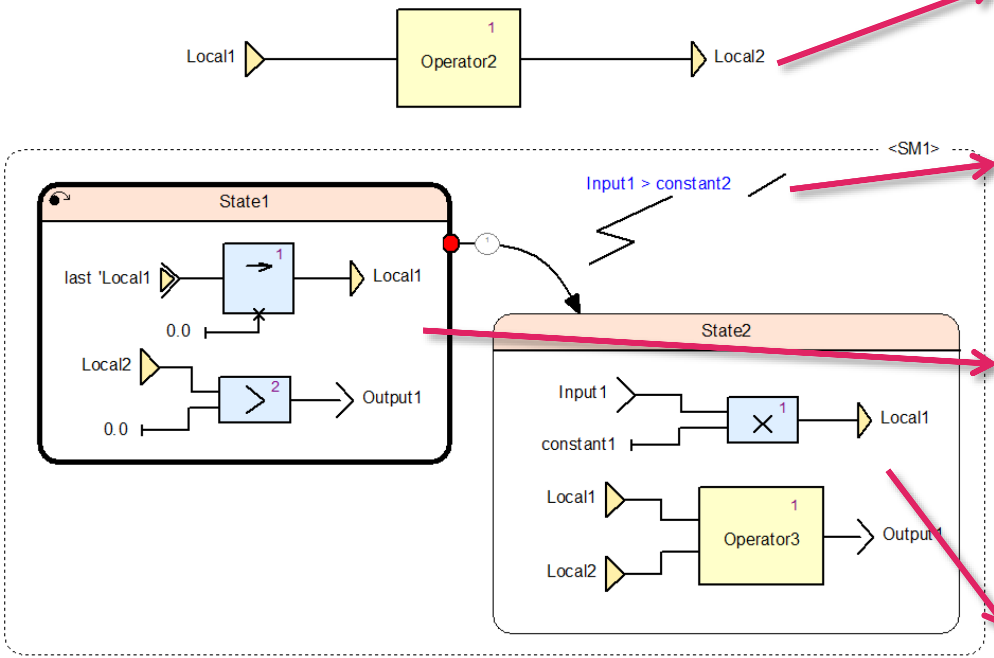
Update function postcondition:




```
(Operator_2.Result_T' (
   State    => Update'Result.State.Operator_2_1_State,
   Output_1 => Update'Result.State.Local_2)
 = Operator_2.Update (
      Old_State => Old_State.Operator_2_1_State,
      Input_1   => Update'Result.State.Local_1)) and
(if Old_State.SM_1 = State_1 then
   (if (Input_1 > Constants.Constant_2) then
      Update'Result.State.SM_1 = State_2
    else
      Update'Result.State.SM_1 = Old_State.SM_1)) and
(if Old_State.SM_1 = State_2 then
   Update'Result.State.SM_1 = Old_State.SM_1) and
(if Update'Result.State.SM_1 = State_1 then
   Update'Result.State.Local_1 =
      (if (Old_State.SM_1 = State_1) and Old_State.Init_1_Evaluated
       then Old_State.Local_1 else 0.0) and
   Update'Result.Output_1 = (Update'Result.State.Local_2 > 0.0) and
   Update'Result.State.Init_1_Evaluated and
   Update'Result.State.Operator_3_1_State =
      Old_State.Operator_3_1_State) and
```

# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS

Update function postcondition:



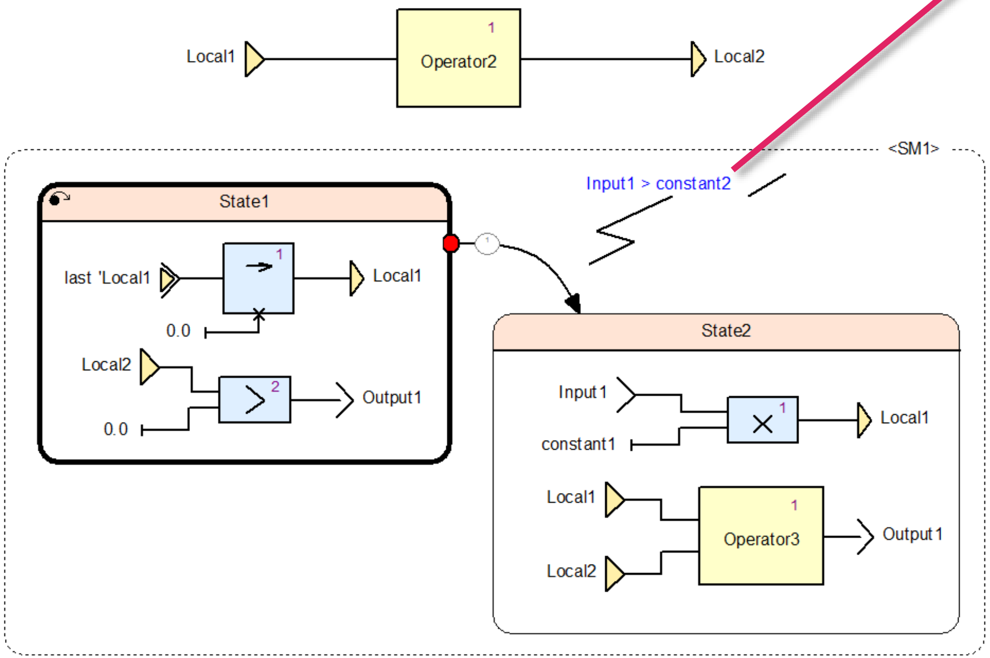

```
(Operator_2.Result_T' (
   State    => Update'Result.State.Operator_2_1_State,
   Output_1 => Update'Result.State.Local_2)
 = Operator_2.Update (
     Old_State => Old_State.Operator_2_1_State,
     Input_1   => Update'Result.State.Local_1)) and
(if Old_State.SM_1 = State_1 then
    (if (Input_1 > Constants.Constant_2) then
      Update'Result.State.SM_1 = State_2
     else
      Update'Result.State.SM_1 = Old_State.SM_1)) and
(if Old_State.SM_1 = State_2 then
   Update'Result.State.SM_1 = Old_State.SM_1) and
(if Update'Result.State.SM_1 = State_1 then
   Update'Result.State.Local_1 =
      (if (Old_State.SM_1 = State_1) and Old_State.Init_1_Evaluated
       then Old_State.Local_1 else 0.0) and
   Update'Result.Output_1 = (Update'Result.State.Local_2 > 0.0) and
   Update'Result.State.Init_1_Evaluated and
   Update'Result.State.Operator_3_1_State =
      Old_State.Operator_3_1_State) and
(if Update'Result.State.SM_1 = State_2 then
   Update'Result.State.Local_1 = (Input_1 * Constants.Constant_1) and
   (Operator_3.Result_T' (
      State    => Update'Result.State.Operator_3_1_State,
      Output_1 => Update'Result.Output_1)
    = Operator_3.Update (
        Old_State => (if Old_State.SM_1 = State_2 then
           Old_State.Operator_3_1_State else Operator_3.Initialise),
        Input_1   => Update'Result.State.Local_1,
        Input_2   => Update'Result.State.Local_2)) and
   Update'Result.State.Init_1_Evaluated =
      Old_State.Init_1_Evaluated));
```

altran

# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS

Update function body:

```
Result.State.SM_1 := (if (Old_State.SM_1 = State_1) then (if
     (Input_1 > Constants.Constant_2) then State_2 else
     Old_State.SM_1) else Old_State.SM_1);
```

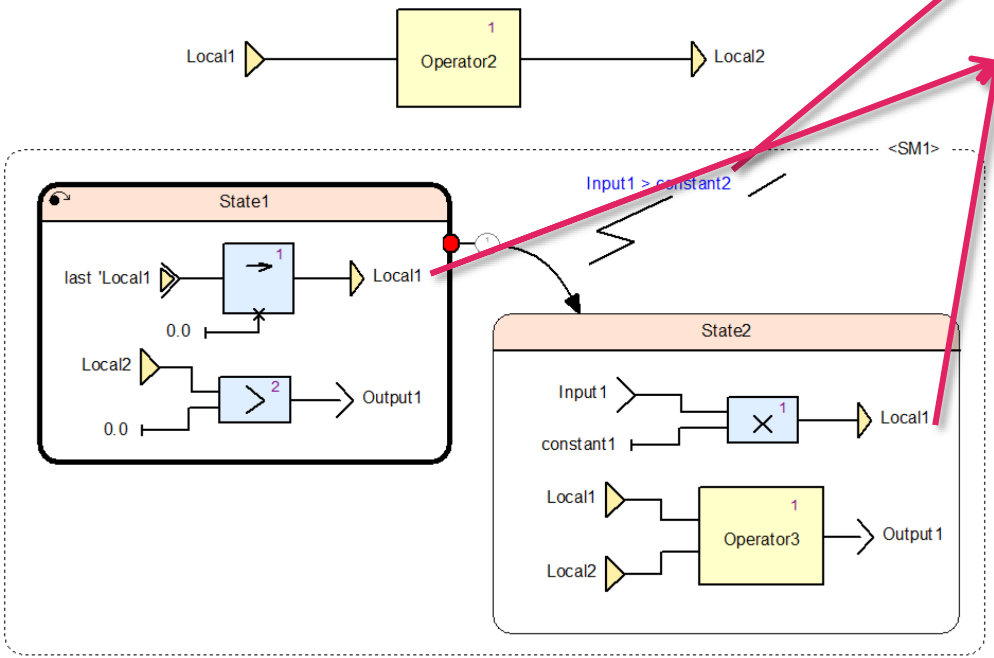



altran

# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS

Update function body:

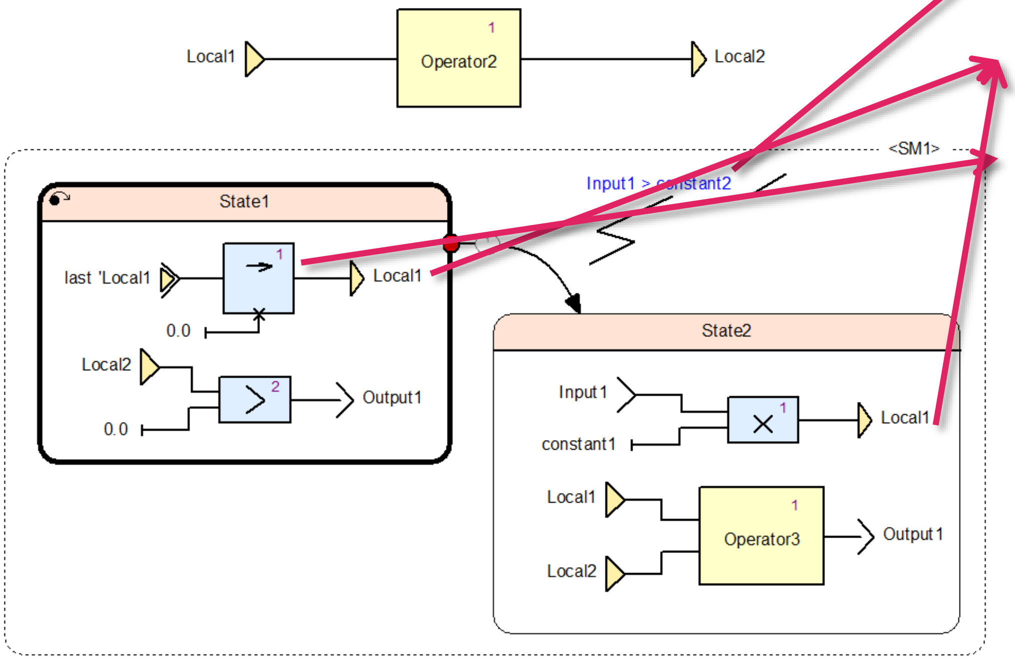
```
Result.State.SM_1 := (if (Old_State.SM_1 = State_1) then (if
      (Input_1 > Constants.Constant_2) then State_2 else
      Old_State.SM_1) else Old_State.SM_1);

Result.State.Local_1 := (if (Result.State.SM_1 = State_1) then (if
      ((Old_State.SM_1 = State_1) and Old_State.Init_1_Evaluated)
      then Old_State.Local_1 else 0.0) else (Input_1 *
      Constants.Constant_1));
```



# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS

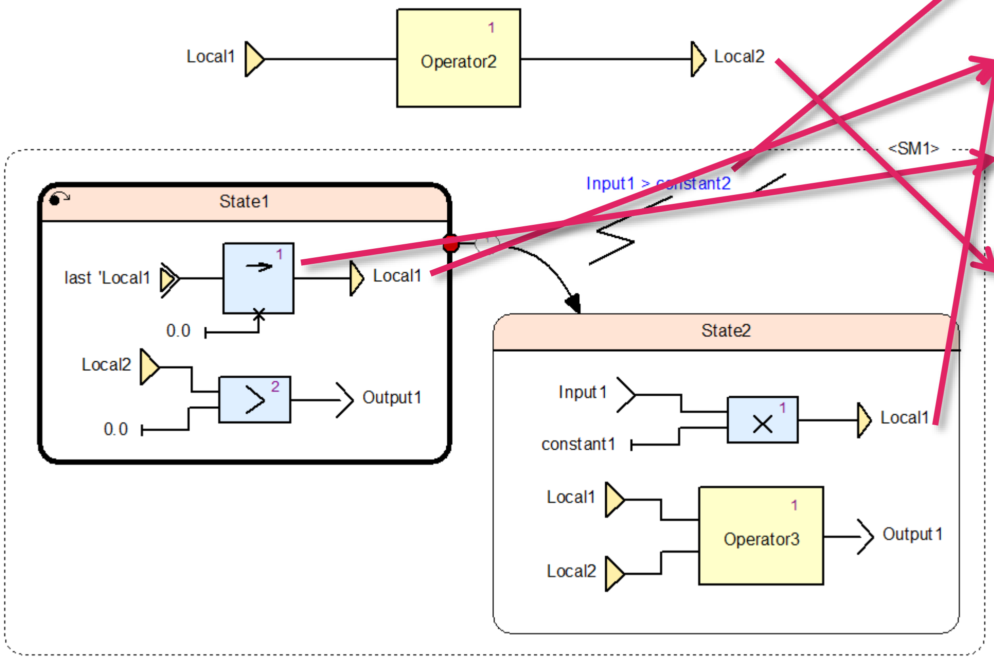
Update function body:

```
Result.State.SM_1 := (if (Old_State.SM_1 = State_1) then (if
      (Input_1 > Constants.Constant_2) then State_2 else
      Old_State.SM_1) else Old_State.SM_1);

Result.State.Local_1 := (if (Result.State.SM_1 = State_1) then (if
      ((Old_State.SM_1 = State_1) and Old_State.Init_1_Evaluated)
      then Old_State.Local_1 else 0.0) else (Input_1 *
      Constants.Constant_1));

Result.State.Init_1_Evaluated := (if (Result.State.SM_1 = State_1)
      then True else Old_State.Init_1_Evaluated);
```



# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS

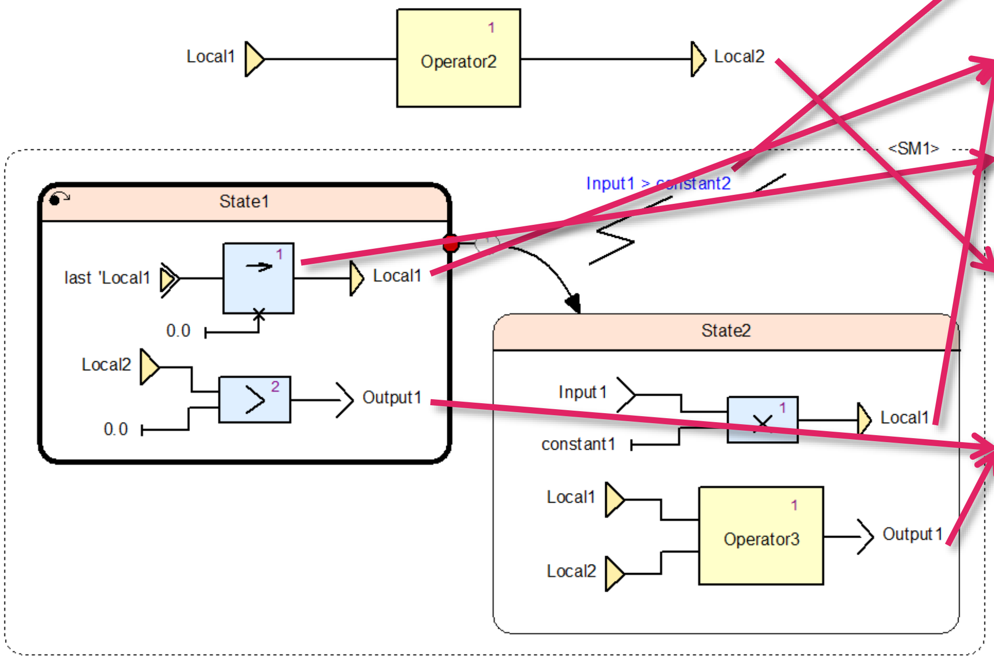
Update function body:




```
Result.State.SM_1 := (if (Old_State.SM_1 = State_1) then (if
      (Input_1 > Constants.Constant_2) then State_2 else
      Old_State.SM_1) else Old_State.SM_1);

Result.State.Local_1 := (if (Result.State.SM_1 = State_1) then (if
      ((Old_State.SM_1 = State_1) and Old_State.Init_1_Evaluated)
      then Old_State.Local_1 else 0.0) else (Input_1 *
      Constants.Constant_1));

Result.State.Init_1_Evaluated := (if (Result.State.SM_1 = State_1)
      then True else Old_State.Init_1_Evaluated);

Result.State.Local_2 := Operator_2.Update (
         Old_State => Old_State.Operator_2_1_State,
         Input_1   => Result.State.Local_1).Output_1;
Result.State.Operator_2_1_State := Operator_2.Update (
         Old_State => Old_State.Operator_2_1_State,
         Input_1   => Result.State.Local_1).State;
```

# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS

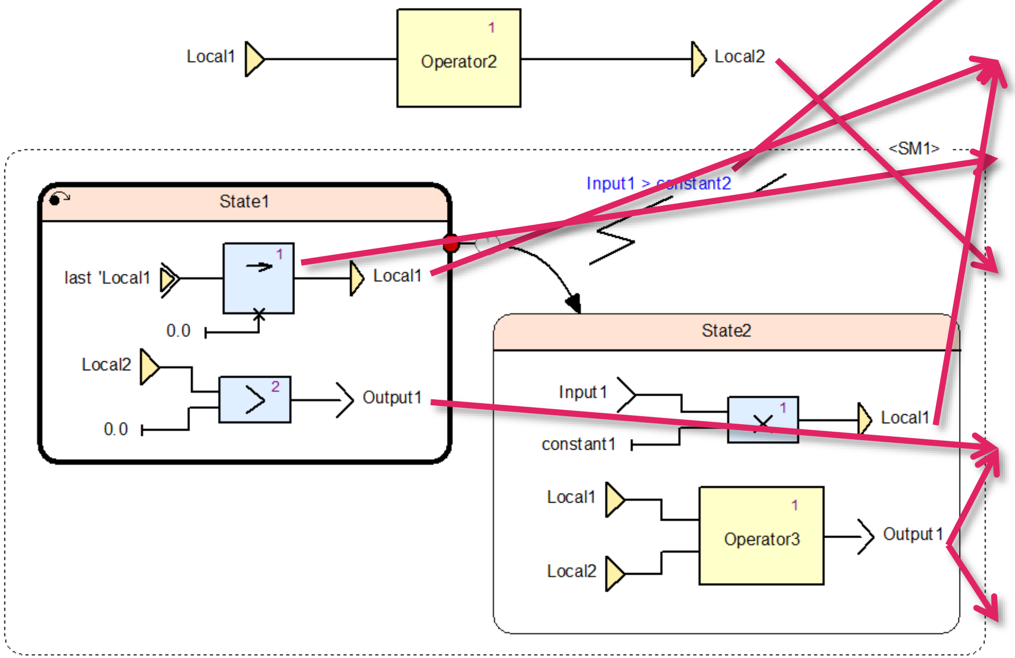
Update function body:




```
Result.State.SM_1 := (if (Old_State.SM_1 = State_1) then (if
      (Input_1 > Constants.Constant_2) then State_2 else
      Old_State.SM_1) else Old_State.SM_1);

Result.State.Local_1 := (if (Result.State.SM_1 = State_1) then (if
      ((Old_State.SM_1 = State_1) and Old_State.Init_1_Evaluated)
      then Old_State.Local_1 else 0.0) else (Input_1 *
      Constants.Constant_1));

Result.State.Init_1_Evaluated := (if (Result.State.SM_1 = State_1)
      then True else Old_State.Init_1_Evaluated);

Result.State.Local_2 := Operator_2.Update (
         Old_State => Old_State.Operator_2_1_State,
         Input_1   => Result.State.Local_1).Output_1;
Result.State.Operator_2_1_State := Operator_2.Update (
         Old_State => Old_State.Operator_2_1_State,
         Input_1   => Result.State.Local_1).State;

Result.Output_1 := (if (Result.State.SM_1 = State_1) then
      (Result.State.Local_2 > 0.0) else Operator_3.Update (
         Old_State => (if Old_State.SM_1 = State_2 then
      Old_State.Operator_3_1_State else Operator_3.Initialise),
         Input_1 => Result.State.Local_1,
         Input_2 => Result.State.Local_2).Output_1);
```

# USE OF CONTRACTS DURING DEVELOPMENT – SCADE REQS

Update function body:



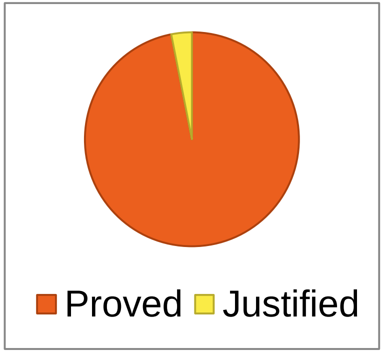

```
Result.State.SM_1 := (if (Old_State.SM_1 = State_1) then (if
     (Input_1 > Constants.Constant_2) then State_2 else
     Old_State.SM_1) else Old_State.SM_1);

Result.State.Local_1 := (if (Result.State.SM_1 = State_1) then (if
     ((Old_State.SM_1 = State_1) and Old_State.Init_1_Evaluated)
     then Old_State.Local_1 else 0.0) else (Input_1 *
     Constants.Constant_1));

Result.State.Init_1_Evaluated := (if (Result.State.SM_1 = State_1)
     then True else Old_State.Init_1_Evaluated);

Result.State.Local_2 := Operator_2.Update (
        Old_State => Old_State.Operator_2_1_State,
        Input_1   => Result.State.Local_1).Output_1;
Result.State.Operator_2_1_State := Operator_2.Update (
        Old_State => Old_State.Operator_2_1_State,
        Input_1   => Result.State.Local_1).State;

Result.Output_1 := (if (Result.State.SM_1 = State_1) then
     (Result.State.Local_2 > 0.0) else Operator_3.Update (
        Old_State => (if Old_State.SM_1 = State_2 then
     Old_State.Operator_3_1_State else Operator_3.Initialise),
        Input_1 => Result.State.Local_1,
        Input_2 => Result.State.Local_2).Output_1);

Result.State.Operator_3_1_State := (if (Result.State.SM_1 = State_2)
     then Operator_3.Update (
        Old_State => Old_State.Operator_3_1_State,
        Input_1   => Result.State.Local_1,
        Input_2   => Result.State.Local_2).State else
     Old_State.Operator_3_1_State);
```

# USE OF CONTRACTS DURING DEVELOPMENT – ENGLISH REQS

- Not all requirements amenable to specification in SCADE e.g.
  - Interface requirements (implemented in abstraction layers of low-level software and hardware)
  - Non-functional requirements (implemented in software and hardware architecture)
- Reverted to our previous style of proof of absence of run-time exceptions, with contracts necessary to support that
- Additional built-in checks added for testing but not proof
  - We didn't prove these because we felt run-time checks were more appropriate than static analysis
  - When interfacing with hardware there is a lot more that can go wrong and there are less solid assumptions on which to base static analysis

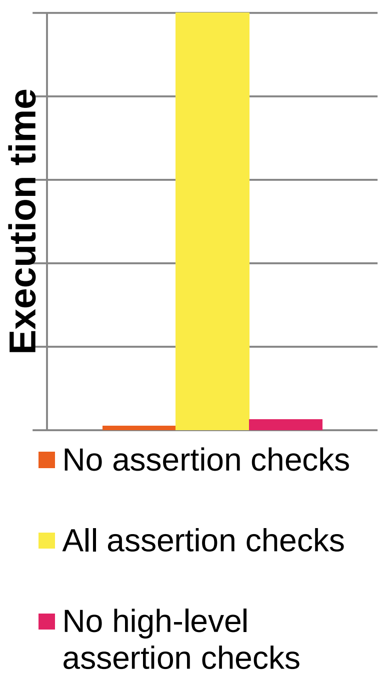
# USE OF CONTRACTS DURING STATIC VERIFICATION

- Proof of implementations against SPARK contracts matching SCADE and of absence of run-time exceptions in all code
- Challenges:
  - Modifications required to SPARK derived from SCADE to support proof
    - Mainly addition of type bounds to types, which was lacking from SCADE
    - We addressed this by manually adding these to the SPARK
  - Management of unproved VCs
    - We didn’t prove 100% of the VCs
    - Engineers made reasonable efforts to prove during development
    - Proof experts worked on reducing these further periodically
    - Static verification report written for releases including rigorous argument for unproved VCs, which was reviewed



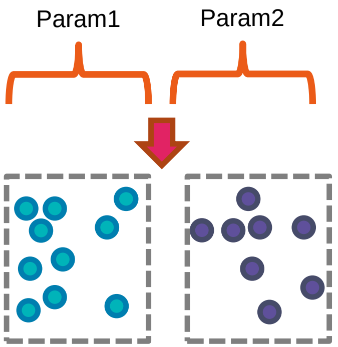
# USE OF CONTRACTS DURING TESTING – ENABLE ASSERTIONS

- We enabled run-time assertion checks, even proved ones
- This was because:
  - Actually, not all VCs are proved (some are justified)
  - It allows us to check the assumptions on which the static analysis is based e.g. no hardware or compiler faults
  - We can take some credit for these in the safety argument
- Run-time cost of checking contracts increases exponentially with call hierarchy
  - Execution time with all run-time checks enabled was over 100 times original
  - Reduced to around 2.5 times original by disabling higher level run-time contract checks

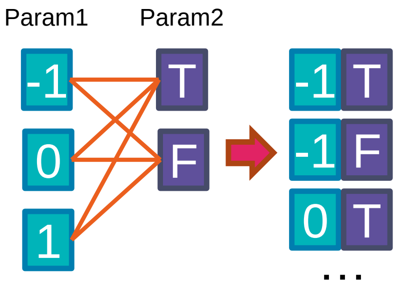


# USE OF CONTRACTS DURING TESTING – DEV MODULE TESTING

- If have built-in assertion checks that capture what you're interested in, all you need to do is generate inputs for tests
- We used a mixture of input generation schemes
- Random input generation
  - Used during production of prototype of system to verify a critical module, in which no defects were ever found
- Cross-product of interesting input values
  - Simple but powerful technique when have assertions
  - E.g. 80,402 interesting input combinations with 1 failure
- Stopped developer testing of proved modules because no defects found




*Random input generation*




*Cross-product of interesting input values*

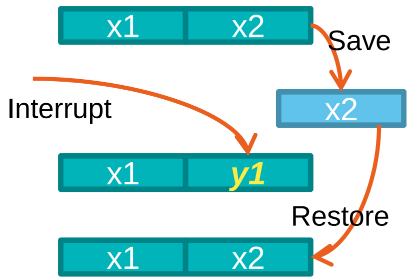
altran

# USE OF CONTRACTS DURING TESTING – IV&V SYSTEM TESTING

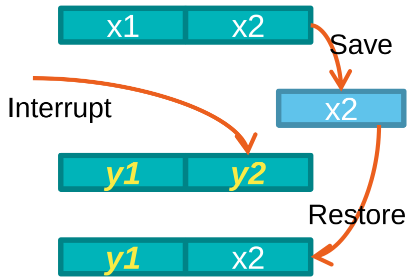
- Independent verification and validation team used a constrained random input generation scheme together with a reference model
- No code faults found in code derived from SCADE requirements
  - We did have some requirements faults, but not many
- There were considerably more requirements and code faults from English requirements
  - The causes typically involved ambiguity in some way
- Where faults in code derived from English requirements were caught by built-in check failures, the faults were much easier to find
  - It was otherwise difficult to debug failures found by the randomly generated tests

# USE OF CONTRACTS DURING TESTING – PROVED CHECKS FAIL

- After an update, various proved postconditions started randomly failing
- The cause was found to be a low-level software fault
  - Register values were being saved before interrupt handlers
  - The registers were 64-bits but only 32-bits were typically used and boot loader was only preserving 32-bits on an interrupt
  - When we used 64-bit floating point operations within interrupt handlers for the first time, if the interrupt handler interrupted a floating point operation then the top 32-bits of the registers could be corrupted
- This showed the ability of run-time assertion checks to catch wider system issues




*Interrupt using 32-bit ops*




*Interrupt using 64-bit ops*

# CONCLUSIONS

- Approach combining heavyweight and lightweight SPARK 2014 contracts and proof and test was usable at highest integrity level
- SPARK contracts can be a good intermediate form in code generation
- Assertions can be effective at finding bugs, even if not proved, when combined with simple test input generation schemes
- Proof works! – no code errors found where full contracts proved
- Formal spec works! – much fewer errors for SCADE than English reqs
- Run-time assertions can help debug failures, particularly in gen. tests
- Enabling of run-time assertion checks worth considering even if proved because can take credit for them and they can find real issues

Acknowledgement: This work was supported by the SECT-AIR project, funded by the Aerospace Technology Institute and Innovate UK, as project number 113099.

altran