Hi-Lite Kick Off

ASTRIUM Space Transportation

Presented by David Lesens

15 July 2010

All the space you need

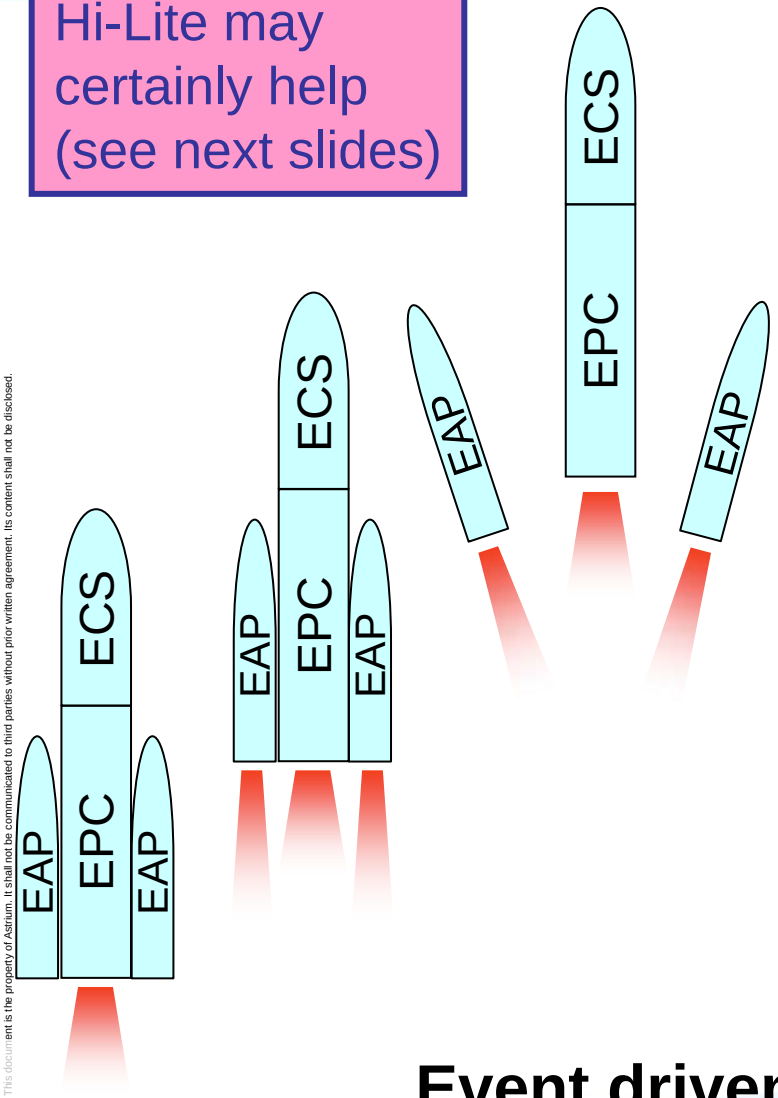
EADS astrium

Overview

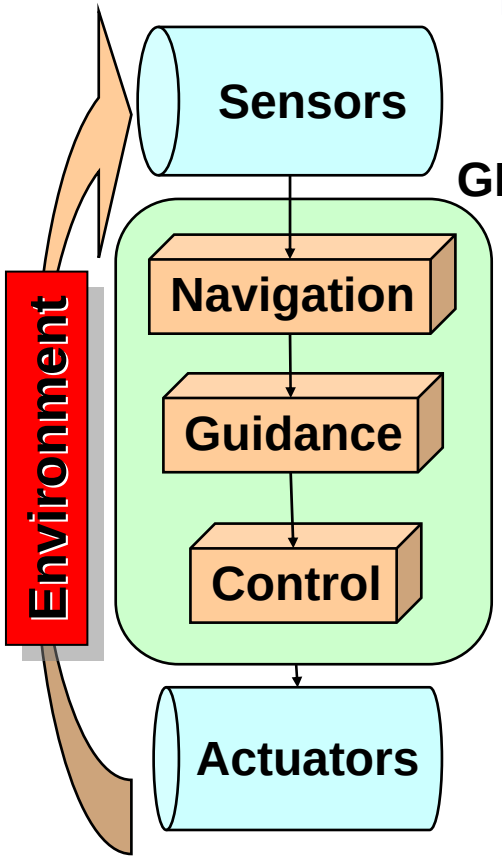
- What is a user requirement?
- The ECSS
  - ECSS at a glance
  - Validation
  - Verification
  - Parametric functions
  - Miscellaneous
- Scenarios of use
- Properties to be proved
- Vectors
- SPARK experimentation first feedback

Software = MVM + GNC

Hi-Lite may certainly help (see next slides)



Can Hi-Lite help? (open question)



- **Acquisition of measurement**
- **Where am I ?**
- **Where shall I go ?**
- **Compute the commands**
- **Send commands to actuators**

Event driven (spacecraft management)

Data flow driven (control / command algorithms)

What is a user requirement? (1/2)

- Feasibility of our systems
  - More autonomy, reliability
- Yesterday
  - Faster, better, cheaper
- Today

  On time, on cost, on quality

What is a user requirement? (2/2)

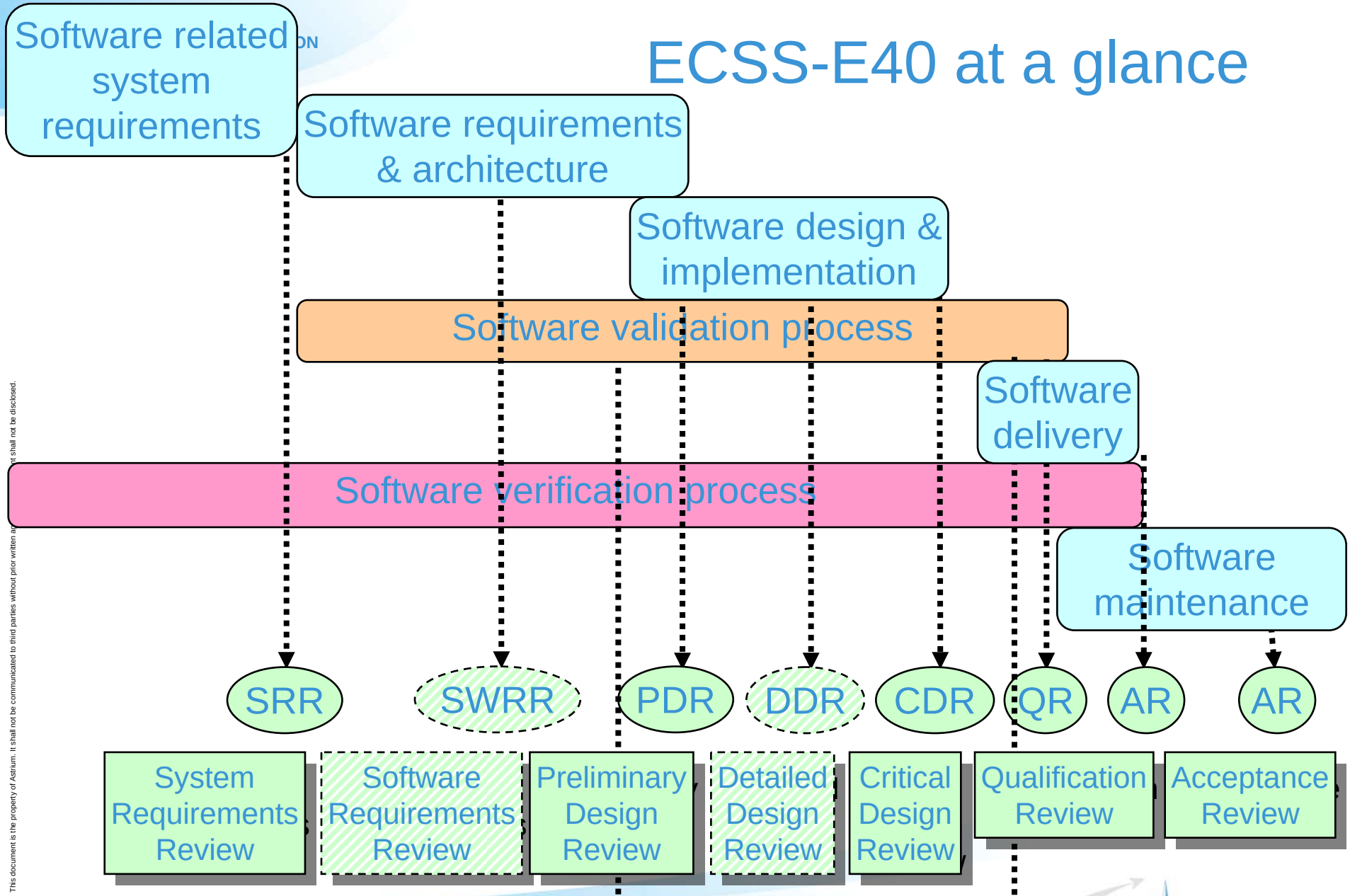
- Some years ago, nobody asks
  - to formally prove the absence of runtime error
  - to formally prove functional properties on Ada code
- Because nobody imagines that possible!

➔ Feel free to propose new “unimaginable” requirements!

Overview

- What is a user requirement?
- The ECSS
  - ECSS at a glance
  - Validation
  - Verification
  - Parametric functions
  - Miscellaneous
- Scenarios of use
- Properties to be proved
- Vectors
- SPARK experimentation first feedback

ECSS-E40 at a glance



Hi-L… Lint integrated wi… Execution

ECSS wording ≠ DO178 wording

- **3.2.44 validation**
  - <software> process to confirm that the requirements baseline functions and performances are correctly and completely implemented in the final product
- **3.2.45 verification**
  - <software> process to confirm that adequate specifications and inputs exist for any activity, and that the outputs of the activities are correct and consistent with the specifications and input

Extracts from the ECSS

- **6.3.4.6**
  - b. The code evaluation shall be performed in parallel with the coding process, in order to provide feedback to the software programmers.
- **6.3.4.5**
  - a. Use of low-level programming languages shall be justified

Is C a low level programming language?

➔ Let us use Ada and Hi-Lite!!

Overview

- What is a user requirement?
- The ECSS
  - ECSS at a glance
  - Validation
  - Verification
  - Parametric functions
  - Miscellaneous
- Scenarios of use
- Properties to be proved
- Vectors
- SPARK experimentation first feedback

3 kinds of tests

- **Unit test**
  - test of individual software unit
- **Integration testing**
  - testing in which software components, hardware components, or both are combined and tested to evaluate the interaction between them
- Validation tests

Some definitions

- **test design**
  - documentation specifying the details of the test approach for a software feature or combination of software features and identifying associated tests

Inputs of following activities or automatic documentation generation

- **test case**
  - set of test inputs, execution conditions and expected results developed for a particular objective such as to exercise a particular program path or to verify compliance with a specified requirement

Formal pre/post-conditions?

- **test procedure**
  - detailed instructions for the set up, operation and evaluation of the results for a given test

The test itself

Verification of the coverage of formal test cases expressed in ALFA by the test procedures

5.5.3.2 Software unit testing

- a. The supplier shall develop and document the test procedures and data for testing each software unit.
- …
- c. The unit test shall exercise
  - 1. code using boundaries at n-1, n, n+1 including looping instructions, while, for and tests that use comparisons
  - …
  - 3. the access of all global variables as specified in the design document
  - 4. out of range values for input data, including values that can cause erroneous results in mathematical functions
  - 5. the software at the limits of its requirements (stress testing)

Inclusion of test procedures documentation in the code

Help to the test cases specification

SPARK verifies it (but does not validate it)

Tests may violate pre-conditions??

Formal proof ensures a full coverage

5.5.4.2 Software integration and testing

- a. The supplier shall integrate the software units and software components, and test them, as the aggregates are developed, in accordance with the integration plan, ensuring that each aggregate satisfies the requirements of the software item and that the software item is integrated at the conclusion of the integration activity

Use of pre and post condition to define the integration correctness

5.6.4.1 Software validation w.r.t the requirements baseline (1/2)

- a. The supplier shall develop and document, for each requirement of the software item in RB (including IRD) , a set of tests, test cases (inputs, outputs, test criteria) and test procedures including:
  - 1. testing against the mission data and scenario specified by the customer
  - 2. testing with stress, boundary, and singular inputs
  - ...
- b. Validation shall be performed by test
- c. If it can be justified that validation by test cannot be performed, validation shall be performed by either analysis, inspection or review of design

See parametric software

Can hi-Lite help? (see next slide)

We can not use formal proof?

Yes, we can
But formal proof can just complete the tests

Some definitions

- **singular input**
  - input corresponding to a singularity of the function

```
if x > 2 then
        …
else
        …
end if;
```

Can Hi-Lite verify a kind of “coverage” of the pre-conditions / partitions ??

➔ **Verification of the definition of the partition x > 2**

- **stress test**
  - test that evaluates a system or software component at or beyond its required capabilities

Tests may violate pre-conditions??
Or SPARK may show that
this requirement is … useless

5.6.4.2 Conducting the validation with respect to the requirements baseline

- a. The validation tests shall be conducted as specified in the output of clause 5.6.4.1
- b. The validation tests shall be "black box", i.e. performed on the final software product to be delivered, without any modification of the code or of the data

Formal proof can not replace validation testing
➔ tests and proof shall be complementary

Regression testing (software)

- Definition
  - selective retesting of a system or component to verify that modifications have not caused unintended effects and that the system or component still complies with its specified requirements
- **6.3.5.15**
  - a. Areas affected by any modification shall be identified and re-tested (regression testing).

Partial automation thanks to formal proof

Overview

- What is a user requirement?
- The ECSS
  - ECSS at a glance
  - Validation
  - Verification
  - Parametric functions
  - Miscellaneous
- Scenarios of use
- Properties to be proved
- Vectors
- SPARK experimentation first feedback

5.8.3.4 Verification of the software detailed design

- a. The supplier shall verify the software detailed design ensuring that:
  - …
  - 5. testing is feasible, by assessing that:
    - (a) commandability and observability features are identified and included in the detailed design in order to prepare the effective testing of the performance requirements
    - (b) computational invariant properties and temporal properties are added within the design
  - …
  - 7. the design is correct with respect to requirements and interfaces, including safety, security, and other critical requirements;
  - …
  - 11. the dynamic features (tasks definition and priorities, synchronization mechanisms, shared resources management) are provided and the real-time choices are justified
  - …

Notion of reachability
SPARK data flow analysis

Idem

Pssible in SPARK?

Idem?

Objective of Hi-Lite

See dedicated slides

5.8.3.5 Verification of code (1/2)

- a. The supplier shall verify the software code ensuring that:
  - …
  - 2. there is internal consistency between software units
  - 3. the code is traceable to design and requirements, testable, correct, and in conformity to software requirements and coding standards
  - 6. the code implements safety, security, and other critical requirements correctly as shown by appropriate methods
  - 7. the effects of run–time errors are controlled;
  - 8. there are no memory leaks
  - 9. numerical protection mechanisms are implemented
- …

Basis of SPARK

See “example of scenario”: Link Test cases / proof

The first objective of formal proof

Linked to the run-time errors

5.8.3.5 Verification of code (2/2)

- …
- d. If it can be justified that the required percentage cannot be achieved by test execution, then analysis, inspection or review of design shall be applied to the non covered code
- …

Proof may sometimes replace tests

- f. The supplier shall verify source code robustness (e.g. resource sharing, division by zero, pointers, run-time errors)
  - AIM: use static analysis for the errors that are difficult to detect at runtime

The first objective of formal proof

5.8.3.6 Verification of software unit testing

- a. The supplier shall verify the unit tests results ensuring that:
  - 1. the unit tests are consistent with detailed design and requirements
  - 2. the unit tests are traceable to software requirements, design and code
  - …
  - 6. test results conform to expected results;
  - …
  - 8. normal termination (i.e. the test end criteria defined in the unit test plan) is achieved
  - …

Executable post-conditions

Links between Test Cases and pre/post-conditions

Can Hi-Lite help?

5.8.3.7 Verification of software integration

- a. The supplier shall verify that the integration has been performed according to the strategy specified in the software integration test plan, and the integration activities ensuring:
  - ...
  - 2. internal consistency;
  - 3. interface testing goals;
  - 4. conformance to expected results

Same than before

5.8.3.8 Verification of software validation with respect to the technical specifications and the requirements baseline

- a. The supplier shall verify the software validation results ensuring that the test requirements, test cases, test specifications, analysis, inspection and review of design cover all software requirements of the technical specification or the requirements baseline

It seems difficult/impossible to represent
software requirements (& validation) by pre/post-conditions
Shall Hi-Lite be restricted to software architectural (& integration)
and detailed design (& unitary testing)

- b. The supplier shall verify the software validation results ensuring conformance to expected results

Coverage (1/2)

- 3.2.2 code coverage
  - percentage of the software that has been executed (covered) by the test suite
- 3.2.10 decision coverage
  - measure of the part of the program within which every point of entry and exit is invoked at least once and every decision has taken “true” and “false” values at least once.
- 3.2.18 modified condition and decision coverage
  - measure of the part of the program within which every point of entry and exit has been invoked at least once, every decision in the program has taken “true” and “false” values at least once, and each condition in a decision has been shown to independently affect that decision’s outcome
- 3.2.35 statement coverage
  - measure of the part of the program within which every executable source code statement has been invoked at least once

Coverage (2/2)

Code coverage versus criticality category	A	B	C	D
Source code statement coverage	100%	100%	AM	AM
Source code decision coverage	100%	100%	AM	AM
Source code modified condition and decision coverage	100%	AM	AM	AM
NOTE: "AM" means that the value is agreed with the customer and measured as per ECSS-Q-ST-80 clause 6.3.5.2.				

Can Hi-Lite help?
Is there a notion of coverage of the proof?

Does a proof coverage correspond
- to a requirement coverage?
  (but can ALFA model high level requirements?)
- to a code coverage?

Overview

- What is a user requirement?
- The ECSS
  - ECSS at a glance
  - Validation
  - Verification
  - Parametric functions
  - Miscellaneous
- Scenarios of use
- Properties to be proved
- Vectors
- SPARK experimentation first feedback

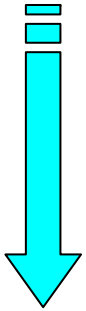
Software customisation

Non customized Flight Software

```
package body DATA is
        TARGETED_ALT := 400_000;
        PAYLOAD_NB := 3;
end DATA;
```

TARGETED_ALT	326.23 km
PAYLOAD_NB	2

Mission data-base



Customized Flight Software for a specific mission

```
package body DATA is
        TARGETED_ALT := 326_2300;
        PAYLOAD_NB := 2;
end DATA;
```

How to prove the correctness of:
- The generic software
- The customized software

Parameterization according to the ECSS

3.2.5 configurable code

- code (source code or executable code) that can be tailored by setting values of parameters
  - This definition covers in particular classes of configurable code obtained by the following configuration means:
    - configuration based on the use of a compilation directive;
    - configuration based on the use of a link directive;
    - configuration performed through a parameter defined in a configuration file;
    - **configuration performed through data defined in a database with impact on the actually executable parts of the software (e.g. parameters defining branch structures that result in the non-execution of existing parts of the code).**

6.3.5.7

Intuitively, Hi-Lite may help

- a. The test coverage of configurable code shall be checked to ensure that the stated requirements are met in each tested configuration

Can we in Hi-Lite replace a constant value by a property on this value?

Example of property of parameterization

- C_X : constant T_FLOAT32 := 5.0;
- C_Y : constant T_FLOAT32 := 10.0;

C_X and C_Y are customizable (their value can be changed by the missionisation tool)

➔ Prove the C_X+2.0 < C_Y

Some definitions

- **deactivated code**
  - code that, although incorporated through correct design and coding, is intended to execute in certain software product configurations only, or in none of them
- **unreachable code**
  - code that cannot be executed due to design or coding error
- **reusability**
  - degree to which a software unit or other work product can be used in more than one computer program or software system

Related problematic

- **5.4.3.7 Reuse of existing software**
  - a. The analysis of the potential reusability of existing software components shall be performed through:
    - 1. identification of the reuse components and models with respect to the functional requirements baseline, and;
    - 2. a quality evaluation of these components

Hi-Lite may help verifying that a reusable building blocks is compatible with an intended use

- **7.3.3 Self-contained information**
  - a. The information related to components developed for reuse in the technical specification, the design justification file, the design definition file and the product assurance file shall be self-contained

Hi-Lite may allow including test cases in the code

Overview

- What is a user requirement?
- The ECSS
  - ECSS at a glance
  - Validation
  - Verification
  - Parametric functions
  - Miscellaneous
- Scenarios of use
- Properties to be proved
- Vectors
- SPARK experimentation first feedback

Real Time

- **5.8.3.11 Schedulability analysis for real-time software**

- **Astrium ST state of the practice:**
  - Rate monotonic Scheduling
  - Harmonic
  - Ravenscar (in progress)

Can Hi-Lite helps?

5.8.3.12 Technical budgets management

- a. As part of the verification of the software requirements and architectural design, the supplier shall estimate the technical budgets including memory size, CPU utilization and the way the deadline are met

Can Hi-Lite helps?

(“I had a dream”)

Numerical Accuracy

- **6.3.3.3**
  - a. For software in which numerical accuracy is relevant to mission success specific rules on design and code shall be defined to ensure that the specified level of accuracy is obtained.
- **7.1.7 Numerical accuracy**
  - a. Numerical accuracy shall be estimated and verified

- Astrium ST approach:
  - Guidelines and code reviews
  - Fluctuat assessment in R&T

Can Hi-Lite helps?

(“I had a dream”)

Overview

- What is a user requirement?
- The ECSS
  - ECSS at a glance
  - Validation
  - Verification
  - Parametric functions
  - Miscellaneous
- Scenarios of use
- Properties to be proved
- Vectors
- SPARK experimentation first feedback

Tests and Contract

- Need to express Test Cases
  - ➔ Pre conditions
  - Post conditions are the expected result
- Example: (© Yannick)
  - function Deliver_Dose return Boolean
    with
    - Partition => (Cash< Price, Changed_My_Mind, Price),
    - Post => Deliver_Dose'Result = Cash > Price and not Changed_My_Mind

Informal Test Cases

- The classical way of working
  - TC1: Test of the case where the cash is not sufficient. The dose is then not delivered
  - TC2: Test of the case where the user changes is mind. The dose is then not delivered
  - TC3: Test the case where the dose is delivered, i.e. the cash is correct and the user does not change its mind
  - TC4: Stress test, the inputs are beyond the required capability of the software. An error is raised

Intuitively, the other cases assume that there is no error

Test cases in Hi-Lite

- Formalised Test Cases
  - TC1 => if Cash >=0 and Cash < Price and not Changed_My_Mind then not Deliver_Dose'Result and not Error;
  - TC2 => if Cash >= Price and Changed_My_Mind then not Deliver_Dose'Result and not Error;
  - TC3 => if Cash >= Price and not Changed_My_Mind then Deliver_Dose'Result and not Error;
  - TC4 => if Cash <0 then not Deliver_Dose'Result and Error;

The absence of error is explicitly written

Advantages of the formalism

1. perform a review of the test cases
   - Review is required by the standard in the scope of the verification activity
2. trace the formal test cases (i.e. the annotations) toward the test plan

Improvement of the test cases

- We can maybe simplify our Test Cases by taking into account the global post-conditions. Something like:
  - Post => Error = Cash <0 and (not Error or not Deliver_Dose'Result);
  - TC1 => if Cash < Price and not Changed_My_Mind then not Deliver_Dose'Result;
  - TC2 => if Cash >= Price and Changed_My_Mind then not Deliver_Dose'Result;
  - TC3 => if Cash >= Price and not Changed_My_Mind then Deliver_Dose'Result;
  - TC4 => if Cash <0 then Error;

The absence of error is factorised

From test cases to test procedures

- Corresponding test procedures
  - (i.e. an implementation or an instantiation of the test cases)
  - TP1: inputs: Cash = 2 and Price = 4 and not Changed_My_Mind; expected outputs: not Deliver_Dose'Result and not Error;
  - TP2: inputs: Cash = 6 and Price = 5 and Changed_My_Mind; expected outputs: not Deliver_Dose'Result and not Error;
  - TP3: inputs: Cash = 5 and Price = 3 and not Changed_My_Mind; expected outputs: Deliver_Dose'Result and not Error;
  - TP4: inputs: Cash = -10 and Price = 2 and not Changed_My_Mind; expected outputs: not Deliver_Dose'Result and Error;

Miscellaneous

- Full traceability between
  - Test Procedures (TPi)
  - Formal annotations
  - Test Cases (TCi)
- Review of all these elements
- Execution of the four TPs

➔ Automatic coverage of the annotations

- (the formal TCs)
- (i.e. a tool will say us after the execution of TP1, that TC1 has be covered)

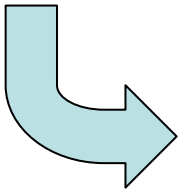
- TC5: Perform the proof of robustness

Scenario of use: Summary

- Writing of informal test cases
- Writing of formal test cases in ALFA
  - Traceability and review
  - Shall we maintain the informal test cases?
- Proofs
  - Shall we prove first or test first?
- Writing of test procedures
- Execution of tests
  - Coverage of the annotations
  - How to handle the complementarity/redundancy between proof/test?

Example from the Tokeneer paper

```
function Sqrt ( X : Integer ) return Integer
with
        pre     => X >= 0
        post    => Sqrt'Result * Sqrt'Result <= X and
                        X < (Sqrt'Result+1) * (Sqrt'Result+1),
        test    => if X = 10 then Sqrt'Result = 3
```



```
function Sqrt ( X : Float ) return Integer
with
        pre     => X >= 0
        post    => Sqrt'Result * Sqrt'Result <= X and
                        X < (Sqrt'Result+1) * (Sqrt'Result+1),
        testcase=> if X < 1 then Sqrt'Result > X,
        testcase=> if X > 1 then Sqrt'Result < X,
        test    => if X = 9.0 then Sqrt'Result = 3.0,
        test    => if X = 0.16 then Sqrt'Result = 0.4;
```

Overview

- What is a user requirement?
- The ECSS
  - ECSS at a glance
  - Validation
  - Verification
  - Parametric functions
  - Miscellaneous
- Scenarios of use
- Properties to be proved
- Vectors
- SPARK experimentation first feedback

General properties

- Absence of run-time error
- Termination
- Real time (meeting the deadlines)
- Mastering of dead code

- Use clauses in Ada
- Overloading of operators
- Use of generic, for instance for matrixes
- Loop invariant like in B

Coherence of a specification

- A specification is said "*coherent*" if it contains no contradiction

The stage release shall be performed on detection of a low threshold of acceleration
The detection of the low threshold of acceleration shall be activated in mode B
The mode B shall be entered at stage release

Completeness of a specification

- A specification is said "*complete*" if it contains exhaustively all the information needed by the developer

The control shall be composed of two blocks A and B
The block A shall be executed during the minor cycle 3
The block B shall be executed during the minor cycle 5
The control shall start on reception of the event X

  - A typical error is to forget specifying that A shall always be executed before B (because it provides some initialisation data)
  - An error can thus occurred if the event X is triggered at minor cycle 4 (B is executed, A is not)
  - This error is not detected if the software is only tested with the event X triggered at minor cycles 1 or 2

Quality of tests

- A test shall not (only) use specific value such as 0 (null value), 1 (neutral value)

- The test cases are formalised by pre and post conditions

➔ Can Hi-Lite help analysing the test quality level?

Interface management

- Reception of telecommand
  - Verification of the correctness of the received data

```
type T_PRO_FAC_DATA is
   record
      DURATIONS : T_CTRL_THR_ACTV_DURATIONS;
      UNUSED    : T_8_BITS;
      CYCLE     : T_NAT8;
   end record;
for T_PRO_FAC_DATA use
   record
      DURATIONS at 0 range 0 .. 127;
      UNUSED    at 0 range 128 .. 135;
      CYCLE     at 0 range 136 .. 143;
   end record;
for T_PRO_FAC_DATA'SIZE use 144;
```

Numerical protection

- Verification that the numerical protections are
  - sufficient
  - mandatory (suppression of useless numerical protections)

Assertions

- In SPARK
  - precondition: i.e. assertion at procedure entrance
  - postcondition: i.e. assertion at procedure exit
- Link between a pre and a post-condition
  - pre X < 0 => post y < 0;
  - pre X > 0 => post y > 0;

  } Links between pre and postconditions Instead of a global precondition and a global postcondition
- Assertion anywhere in the code
  - ASSERT X < Y (after a complex computation)
- Global assertion
  - GLOBAL ASSERT X < Y (at any location in a package)

Standardisation

- Of annotations
- of VC

➔ To be included in ADA 2012?

Automatic Test Generation

- From a formal TC

Overview

- What is a user requirement?
- The ECSS
  - ECSS at a glance
  - Validation
  - Verification
  - Parametric functions
  - Miscellaneous
- Scenarios of use
- Properties to be proved
- Vectors
- SPARK experimentation first feedback

Discussion on vectors

- AdaCore
  - We think that most of our customers in the critical embedded software market will not really need to remove elements from containers
  - They will use containers to define an initial configuration, by adding a bunch of element in the containers, and after that they will only read values in these containers
- Astrium: All cases can be envisaged
  - A list contains a kind of configuration, i.e. which is mainly read
  - Some updates may be performed from time to time
    - (by suppressing or adding some elements from/in the list)
  - ➔ Proof of classical properties such as
    - "for any element in the list, a property is satisfied", or
    - "if it exists an element in the list such as ..., then ..."

Example 1/3: Spacecraft monitoring

- A list stores the set of currently enabled monitoring
  - The temperature of the system is not too low
  - The velocity of a vehicle is not to high
  - etc
- Needed Verification / Validation
  - If no monitoring detects failure, then no recovery is triggered
  - If some monitoring detect failure, then the recovery with the highest priority is triggered

Example 2/3: Telemetry

- Telemetry events are stored in FIFO
- Needed Verification / Validation
  - FIFO sizes (static sizes for critical embedded software) have been correctly set
  - If a buffer is full, then a warning is raised
- Extension of FIFO
  - Satisfaction of an order of emission

Example 3/3: Monitoring of the system state

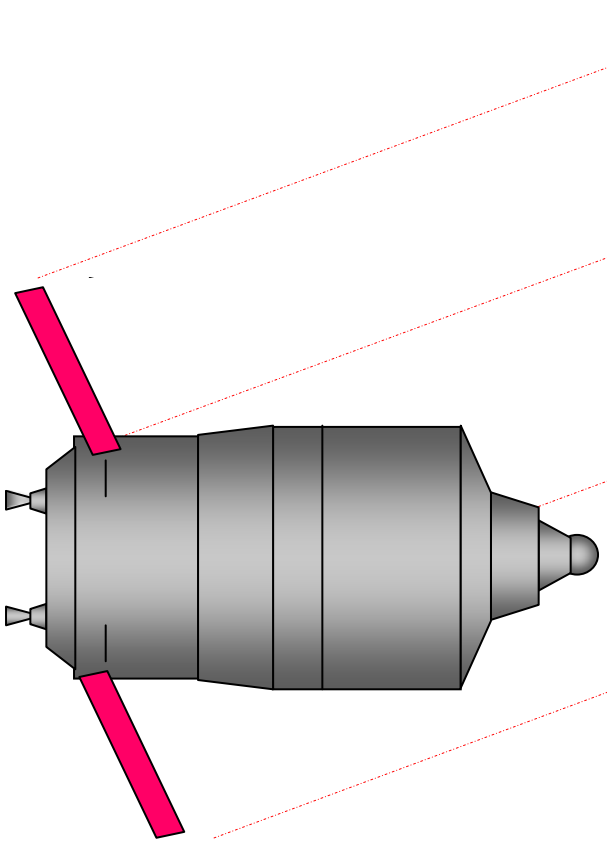
- In a list at most n elements are in a given state
  - For instance, to verify that we have no more than 4 heaters active at the same time in system

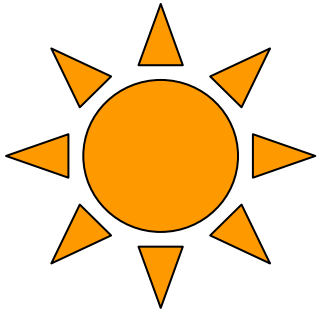
    (because it is the maximal number supported by the power subsystem)

Overview

- What is a user requirement?
- The ECSS
  - ECSS at a glance
  - Validation
  - Verification
  - Parametric functions
  - Miscellaneous
- Scenarios of use
- Properties to be proved
- Vectors
- SPARK experimentation first feedback

ATV Solar Generation System





- **Numerical algorithms**

The Flight Application Software turns the solar wings toward the sun in order to optimize the energy

Approach

- Work in progress
- Sparkification of an existing Ada code
  - (not recommended!)
- First assessment of SPARK
  - To help the writing of user requirements (T2.1)
- During T7.2, a case study will be developed
  - from scratch
  - in Hi-Lite (not in SPARK)

Mathematical Library: Specification

- Overload of operators is forbidden

```
-------------------------------------------------------------------------
-- Float simple precision
-------------------------------------------------------------------------

function SIGN32     ( X       : in T_FLOAT_32)     return T_SIGN_INT32;
function MAX32      ( X1, X2  : in T_FLOAT_32 )    return T_FLOAT_32;
function Sqrt32     ( X       : T_FLOAT_32)        return T_FLOAT_32;
function Sin32      ( X       : T_FLOAT_32)        return T_FLOAT_32;

-------------------------------------------------------------------------
-- Float double precision
-------------------------------------------------------------------------

function SIGN64     ( X       : in T_FLOAT_64)     return T_SIGN_INT32;
function MAX64      ( X1, X2  : in T_FLOAT_64 )    return T_FLOAT_64;
function Sqrt64     ( X       : T_FLOAT_64)        return T_FLOAT_64;
function Sin64      ( X       : T_FLOAT_64)        return T_FLOAT_64;
```

Mathematical Library: Body

- The trigonometric functions are not compatible with SPARK Ada
  - ➔ “Hide” directive

```
with Ada.Numerics.Generic_Elementary_Functions;

package body ML
is

   --# hide ML

end ML;
```

Mathematical Library: Future work

- Addition of pre and post conditions

```
function SIGN32      ( X          : in T_FLOAT_32)     return T_SIGN_INT32;

Pre-condition:       X > -10.0 and
                     X < 10.0
Post-condition:      Result  >= -1 and
                     Result <= -1
```

Specification

- Highlighting of global variables

```
package B6
 --# own SOLAR_VECTOR;
 --# initializes SOLAR_VECTOR;
is
procedure S_SGS_B6_ELAB_SOL_WING_CMD_ANGLE (
                        FRMS_SUN_DIR_IN_ATV_FRAME       : in T_FRMS_VECTOR;
                        YSUN_OCF                        : in ML.T_FLOAT_32;
                        GNC_TO_SGS_G1A_ACTIVE           : in Boolean;
                        SGS_THETA_CONS_LIMIT_COEFF0     : in ML.T_FLOAT_32;
                        SGS_THETA_CONS_LIMIT_COEFF1     : in ML.T_FLOAT_32;
                        SGS_MOTOR_BOARD_ANGLE           : out T_SGS_MOTOR_BOARD_ANGLE
                                );
--# global in out SOLAR_VECTOR;
--# derives SGS_MOTOR_BOARD_ANGLE from
--#                       SOLAR_VECTOR,
--#                       FRMS_SUN_DIR_IN_ATV_FRAME,
--#                       YSUN_OCF,
--#                       GNC_TO_SGS_G1A_ACTIVE,
--#                       SGS_THETA_CONS_LIMIT_COEFF0,
--#                       SGS_THETA_CONS_LIMIT_COEFF1 &
--#       SOLAR_VECTOR from *, FRMS_SUN_DIR_IN_ATV_FRAME;
```

Body

- "use" clause forbidden

```
with ML;
use ML;

XSUN : T_FLOAT_32;

THETA_CONS_BOARD_MOTOR4 := ARCTAN (
      ((YSUN * SIN (C_CONV_SGS_PHI) -
      ZSUN * COS (C_CONV_SGS_PHI)) / XSUN));
```

```
with ML;
use type ML.T_FLOAT_32;
use type ML.T_INT32;

 XSUN : ML.T_FLOAT_32;

THETA_CONS_BOARD_MOTOR4 := ML.ARCTAN32 (
      ((YSUN * ML.SIN32 (C_CONV_SGS_PHI) -
      ZSUN * ML.COS32 (C_CONV_SGS_PHI)) / XSUN));
```

SPARK: error messages (1/2)

```
SGS_MOTOR_BOARD_ANGLE = T_SGS_MOTOR_BOARD_ANGLE'(
                        THETA_CONS_BOARD_MOTOR1,
                        THETA_CONS_BOARD_MOTOR2,
                        THETA_CONS_BOARD_MOTOR3,
                        THETA_CONS_BOARD_MOTOR4 );
```

SPARK Syntax Error SIMPLE_NAME cannot be followed by “=“ here

GNAT “=“ should be “:=“

```
SGS_MOTOR_BOARD_ANGLE := T_SGS_MOTOR_BOARD_ANGLE'(
                         THETA_CONS_BOARD_MOTOR1,
                         THETA_CONS_BOARD_MOTOR2,
                         THETA_CONS_BOARD_MOTOR3,
                         THETA_CONS_BOARD_MOTOR4 );
```

SPARK: error messages (2/2)

```
C_INIT_SOLAR_VECTOR : constant T_VECT_3_FLOAT_32 := (
                                                    1 => - 1.0,
                                                    2 => 0.0,
                                                    3 => 0.0);
```

SPARK

Syntax Error: No complete SIMPLE_EXPRESSION can be followed by "=>" here

```
C_INIT_SOLAR_VECTOR : constant T_VECT_3_FLOAT_32 := T_VECT_3_FLOAT_32'(
                                                    1 => - 1.0,
                                                    2 => 0.0,
                                                    3 => 0.0);
```

Side effects

function asin_32(X)

- Pre condition: -1 <= X <= 1
  - ➔ can it be proved?

  If not implementation of a numerical protection and of a warning mechanism
- If false ➔ raise a warning (posted in a specific buffer)
  - ➔ side effect

  May imply an indeterminism
- But we don't care!!

Generic functions

```
typedef ML::Matrix<int,2,3> Mat2x3;
typedef ML::Matrix<int,3,2> Mat3x2;
```

Very simple in C++

```
package MAT_2_3_FLOAT_32 is new ELEMENTARY_MATRIX(
                                T_G_FLOAT =>ML.T_FLOAT_32,
                                T_G_ROW_INDEX => ML.T_2_RANGE,
                                T_G_COLUMN_INDEX => ML.T_3_RANGE);
```

```
package MAT_3_2_FLOAT_32 is new ELEMENTARY_MATRIX(
                                T_G_FLOAT =>ML.T_FLOAT_32,
                                T_G_ROW_INDEX => ML.T_3_RANGE,
                                T_G_COLUMN_INDEX => ML.T_2_RANGE);
```

```
...
```

Quite complex in Ada

SPARK: First discovery

Experimentation on a simple algorithm

- Feasibility
  - Mathematical library shall be “hidden”
  - Algorithm can be described in SPARK
- Advantages
  - Highlight global variables
  - Suppress any ambiguity of renaming
  - Modular application
- Drawbacks
  - “Heavier” description
- Future works
  - Increase the scope of the assessment
  - Use the “simplifier”

Discussion…