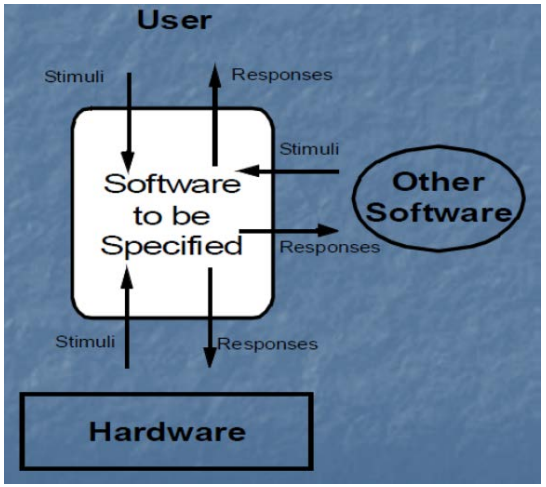
MATTS –
Model-Based Testing of Spacecraft On-board Software

Final Presentation

2015-06-02

The starting point

- Previous studies performed by DNV GL for ESA shows that transfer of a set of software requirements originally written in prose into different forms of formal models, implies valuable early stage validation, in particular related to evaluating the completeness of requirements.
- The previous studies have also shown that the formalisation process is a labour intensive exercise, and therefore the model should be used for more than specification, verification and validation of requirements, to regain costs.
- Therefore this project has investigated formalisation through the Sequence Based Specification (SBS) methodology in combination with Model Based Testing (MBT).



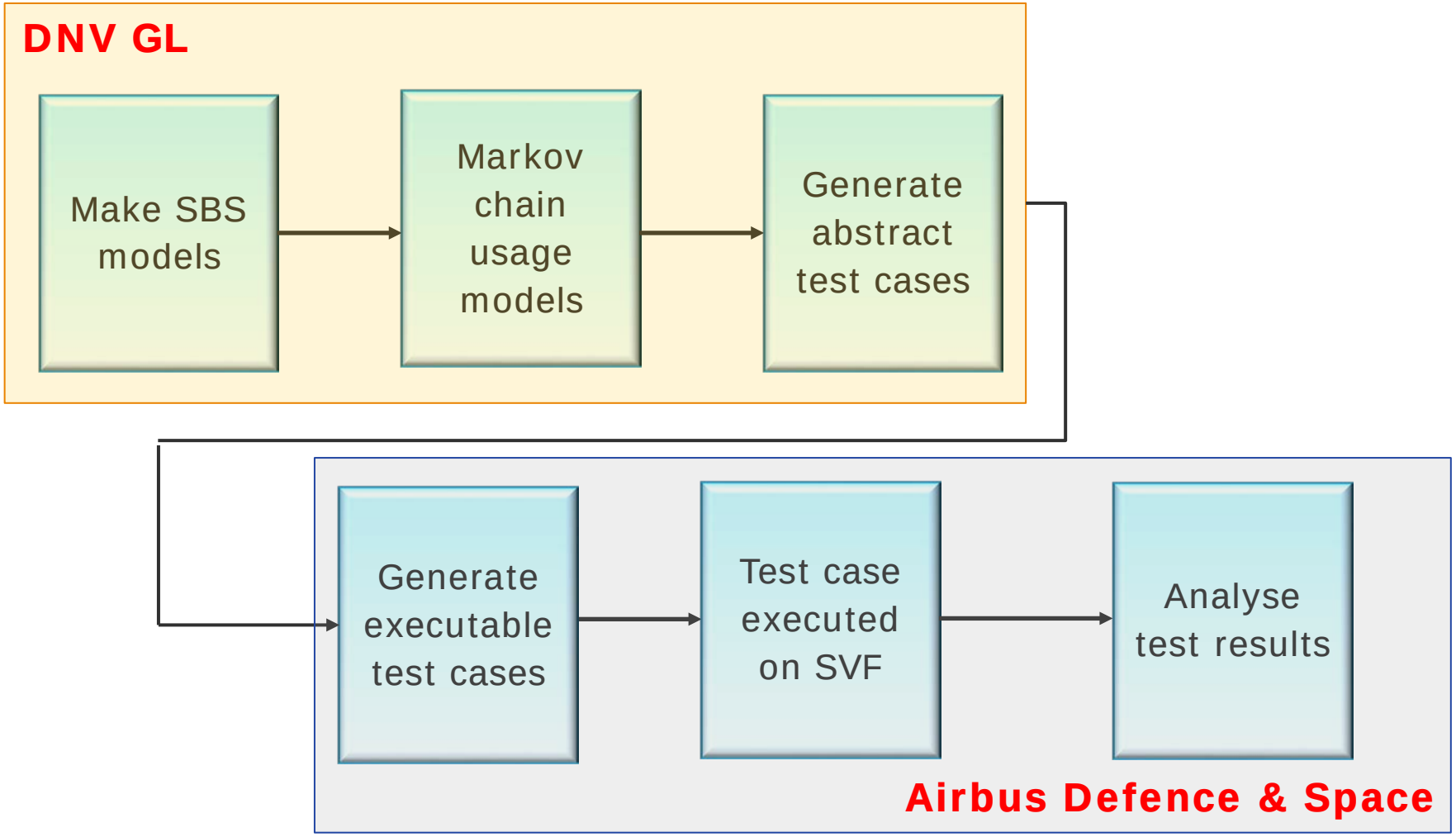
The team

- *Jingyue.Li and Jing.Xie*
  - *DNV GL Strategic Research and Innovation, Høvik, Norway*
- Hans-Juergen.Herpel
  - *Airbus Defence and Space, Friedrichshafen, Germany*
- *Sabine Krueger*
  - *Systems Analysis and Consulting, München, Germany*

- Bengt.Solheimdal.Johansen and Kenneth.Kvinnesland
  - *Security and Information Risk Management DNV GL, Høvik, Norway*

- *Pedro Barrios*
  - *ESA Technical Officer*

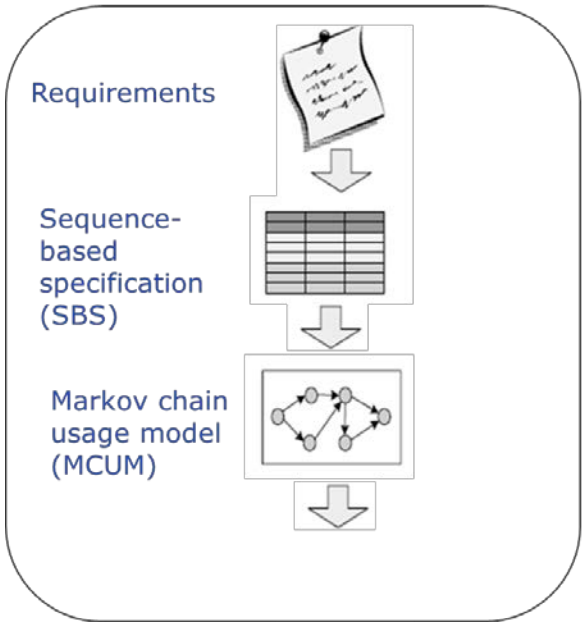
Project Activities and Split of Work



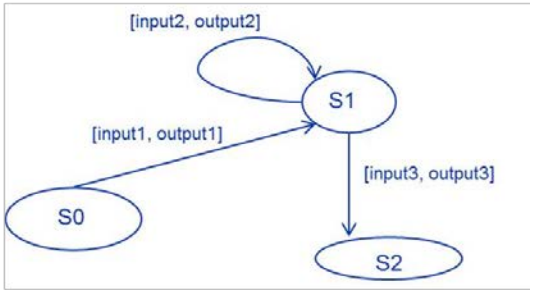
SBS – Introduction

What techniques does the SBS use?

- Sequence enumeration for iteratively deriving a formal specification from requirements written in natural language
- Sequence abstraction for controlling the growth of the enumeration process



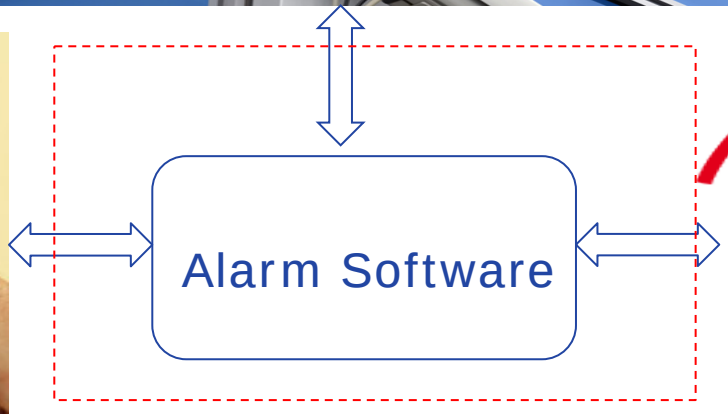
Markov chain usage model is a finite-state machine whose output values are determined both by its current state and the current inputs.)



What does the SBS provide?

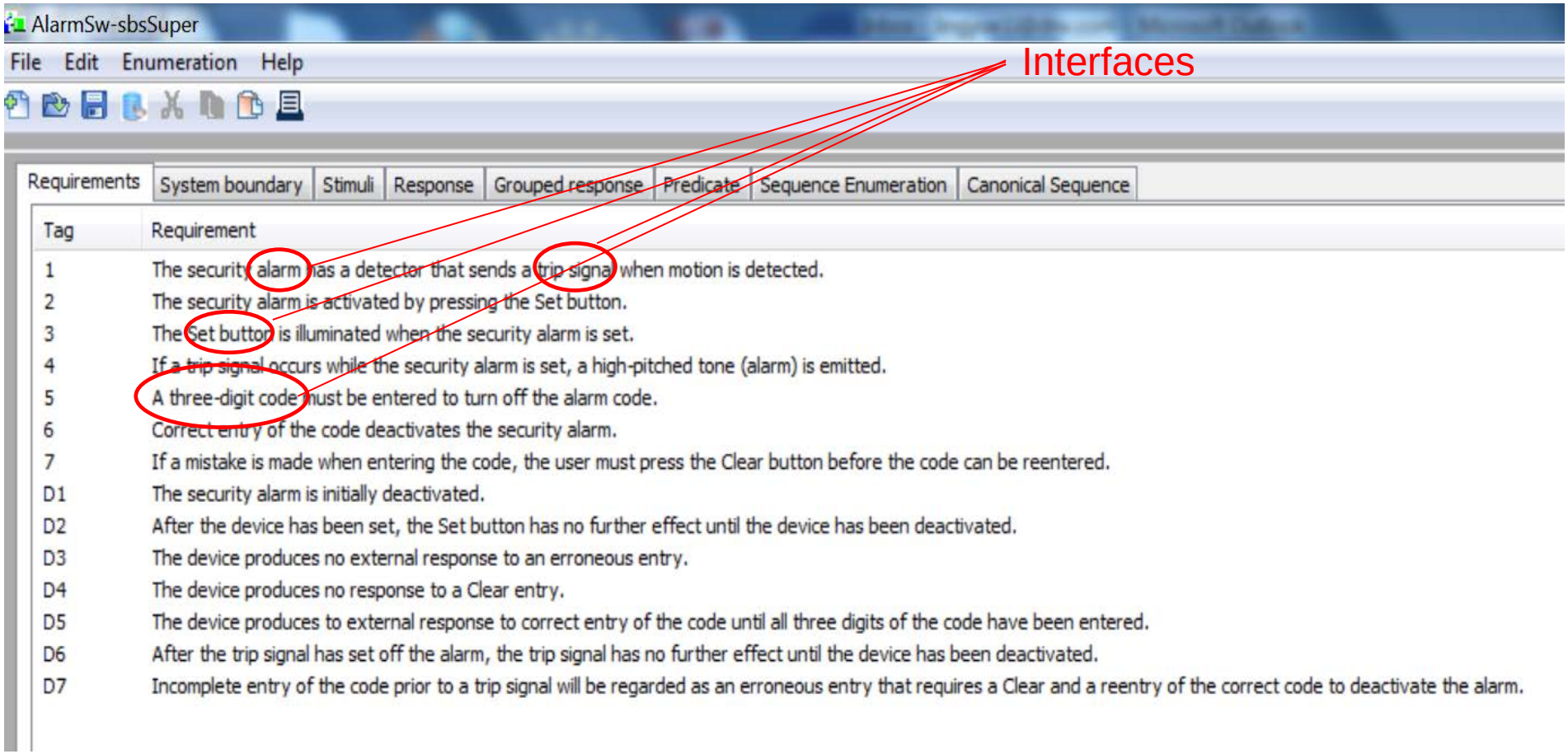
- A systematic way to explore and discover the intended system behaviour in consistent, complete and traceably correct manner

Example: Alarm software



SBS process executable steps

- **Alarm software specification**



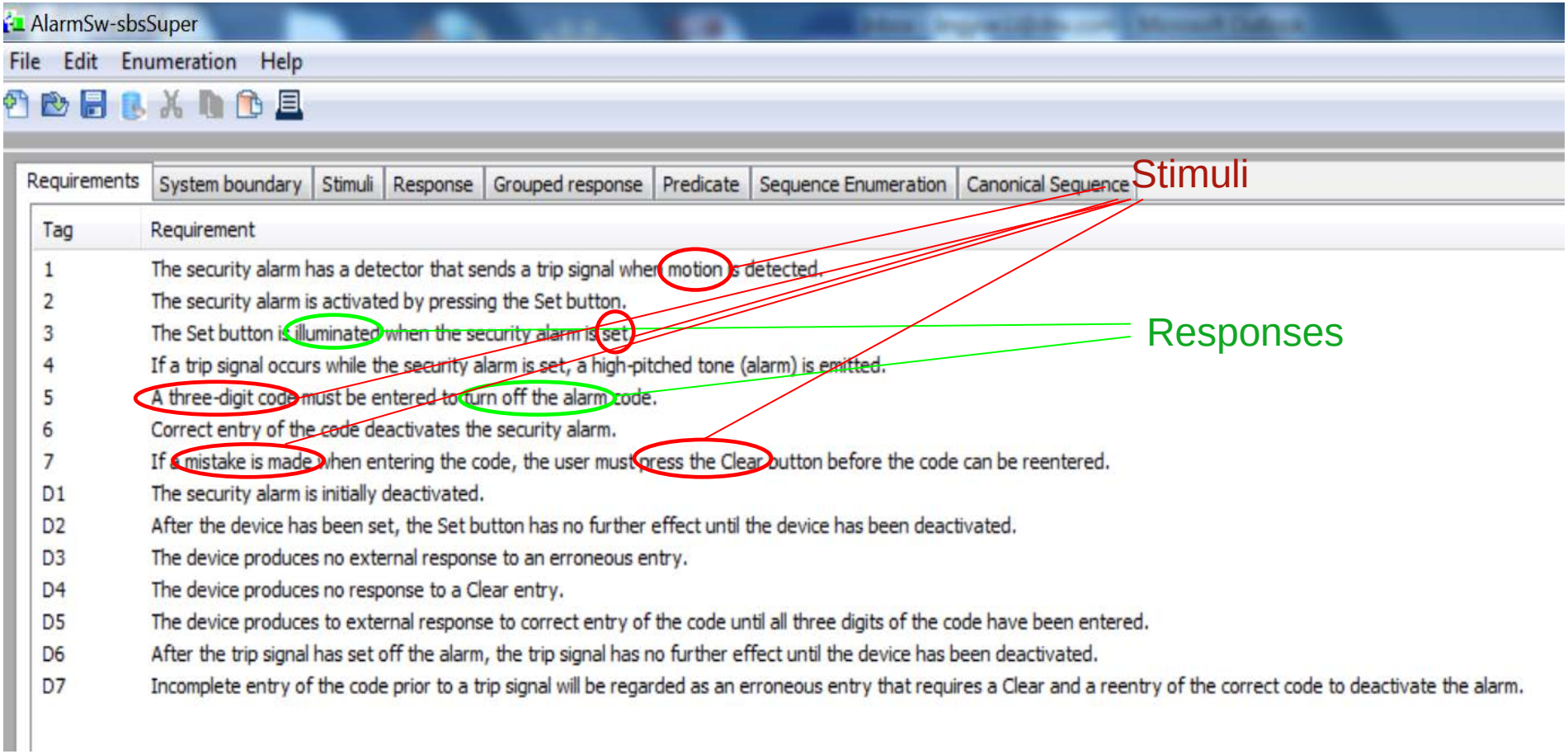
SBS process – Step 1

Step 1. Identify the boundary of the system under modelling and define human / software / hardware interfaces

Requirements | System boundary | Stimuli | Response | Grouped response | Predicate | Sequence Enumeration | Canonical Sequence

Interface	Interface description	Requirement trace
Alarm	The alarm speaker used to generate the alarm sound.	[4]
Key Pad	The key pad where the user enters the code and arms ...	[7,6,5,2]
Trip Wire	The detector that sets off the alarm, if armed.	[4,1]

Alarm software specification



SBS process – Step 2 & Step 3

Step 2: Identify all possible stimuli

Requirements | System boundary | **Stimuli** | Response | Grouped response | Predicate | Sequence Enumeration | Canonical Sequence

Stimuli Name	Stimuli description	Source	Requirement trace
BadDigit	The user has entered a number that is not part of the alarm disarming code.	[Key Pad]	[7]
Clear	The user has pressed the Clear button on the keypad.	[Key Pad]	[7]
GoodDigit	The user has entered the correct next digit of the alarm disarming code.	[Key Pad]	[6,5]
Set	The user has pressed the Set button on the keypad.	[Key Pad]	[2]
Trip	Something has tripped the alarm detector.	[Trip Wire]	[1]

Step 3: Identify all possible system responses

Requirements | System boundary | Stimuli | **Response** | Grouped response | Predicate | Sequence Enumeration | Canonical Sequence

Response Name	Response description	Destination	Requirement trace
Alarm Off	The high-pitched alarm sound has been turned off.	[Alarm]	[5]
Alarm On	The high-pitched alarm sound has been activated.	[Alarm]	[4]
Light Off	The Set button light has been turned off.	[Key Pad]	[6]
Light On	The Set button light has been turned on.	[Key Pad]	[3]
Null	No response	[]	[]

SBS process – Step 4

Step 4. Enumerate sequences of stimuli and assign responses to each sequence

Requirements | System boundary | Stimuli | Response | Grouped response | Predicate | **Sequence Enumeration** | Canonical Sequence

Length	Prefix	Stimulus	Predicate	Predicate value	Response	Legal	Equivalence	Requirement Trac
1	Lambda	BadDigit				No		[D1]
1	Lambda	Clear				No		[D1]
1	Lambda	GoodDigit				No		[D1]
1	Lambda	Set			[Light On]	Yes	---	[2,3]
1	Lambda	Trip				No		[D1]
2	Set	BadDigit			[Null]	Yes	---	[D3]
2	Set	Clear			[Null]	Yes	Set	[D4]
2	Set	GoodDigit			[Null]	Yes	---	[D5]
2	Set	Set			[Null]	Yes	Set	[D2]
2	Set	Trip			[Alarm On]	Yes	---	[1,4]
3	Set.BadDigit	BadDigit			[Null]	Yes	Set.BadDigit	[D3]
3	Set.BadDigit	Clear			[Null]	Yes	Set	[D4,7]
3	Set.BadDigit	GoodDigit				No		[7]
3	Set.BadDigit	Set			[Null]	Yes	Set.BadDigit	[D2]
3	Set.BadDigit	Trip			[Alarm On]	Yes	---	[1,4]
3	Set.GoodDigit	BadDigit			[Null]	Yes	Set.BadDigit	[D3]
3	Set.GoodDigit	Clear			[Null]	Yes	Set	[D4]
3	Set.GoodDigit	GoodDigit			[Null]	Yes	---	[D5]
3	Set.GoodDigit	Set			[Null]	Yes	Set.GoodDigit	[D2]
3	Set.GoodDigit	Trip			[Alarm On]	Yes	Set.BadDigit.Trip	[D7]
3	Set.Trip	BadDigit			[Null]	Yes	Set.BadDigit.Trip	[D3]

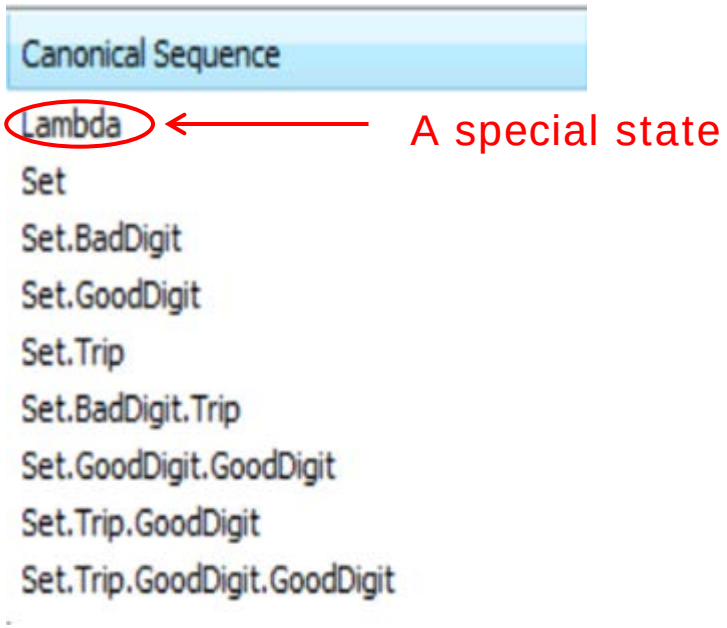
SBS process – Step 4 (Cont.)

Requirements | System boundary | Stimuli | Response | Grouped response | Predicate | **Sequence Enumeration** | Canonical Sequence

Length	Prefix	Stimulus	Predicate	Predicate value	Response	Legal	Equivalence	Requirement Trace
3	Set.Trip	Clear			[Null]	Yes	Set.Trip	[D4]
3	Set.Trip	GoodDigit			[Null]	Yes	---	[D5]
3	Set.Trip	Set			[Null]	Yes	Set.Trip	[D2]
3	Set.Trip	Trip			[Null]	Yes	Set.Trip	[D6]
4	Set.BadDigit.Trip	BadDigit			[Null]	Yes	Set.BadDigit.Trip	[D3]
4	Set.BadDigit.Trip	Clear			[Null]	Yes	Set.Trip	[D4,7]
4	Set.BadDigit.Trip	GoodDigit				No		[]
4	Set.BadDigit.Trip	Set			[Null]	Yes	Set.BadDigit.Trip	[D2]
4	Set.BadDigit.Trip	Trip			[Null]	Yes	Set.BadDigit.Trip	[D6]
4	Set.GoodDigit.GoodDigit	BadDigit			[Null]	Yes	Set.BadDigit	[D3]
4	Set.GoodDigit.GoodDigit	Clear			[Null]	Yes	Set	[D3]
4	Set.GoodDigit.GoodDigit	GoodDigit				No		[6,5]
4	Set.GoodDigit.GoodDigit	Set			[Null]	Yes	Set.GoodDigit....	[D2]
4	Set.GoodDigit.GoodDigit	Trip			[Alarm On]	Yes	Set.BadDigit.Trip	[D7,4]
4	Set.Trip.GoodDigit	BadDigit			[Null]	Yes	Set.BadDigit.Trip	[D3]
4	Set.Trip.GoodDigit	Clear			[Null]	Yes	Set.Trip	[D4]
4	Set.Trip.GoodDigit	GoodDigit			[Null]	Yes	---	[D5]
4	Set.Trip.GoodDigit	Set			[Null]	Yes	Set.Trip.Good...	[D2]
4	Set.Trip.GoodDigit	Trip			[Null]	Yes	Set.Trip.Good...	[D6]
5	Set.Trip.GoodDigit.GoodDigit	BadDigit			[Null]	Yes	Set.BadDigit.Trip	[D3]
5	Set.Trip.GoodDigit.GoodDigit	Clear			[Null]	Yes	Set.Trip	[D4]
5	Set.Trip.GoodDigit.GoodDigit	GoodDigit			[Light Off...	Yes	Lambda	[3,6,5]
5	Set.Trip.GoodDigit.GoodDigit	Set			[Null]	Yes	Set.Trip.Good...	[D2]
5	Set.Trip.GoodDigit.GoodDigit	Trip			[Null]	Yes	Set.Trip.Good...	[D6]

SBS process – Step 5

Step 5. Perform canonical sequence analysis



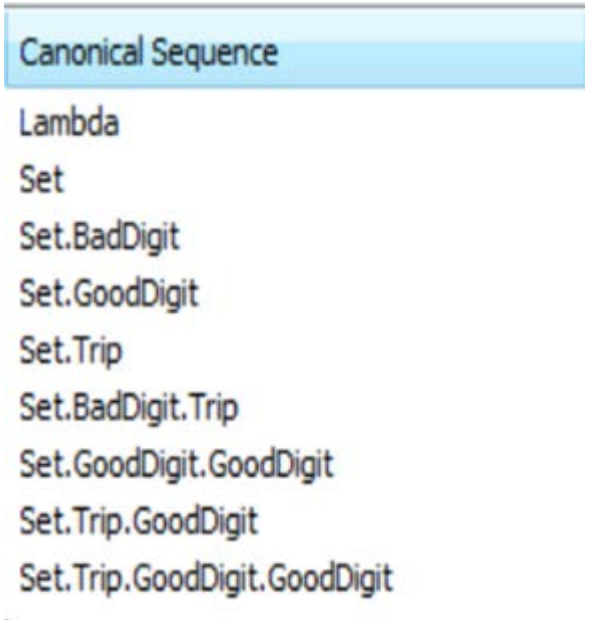
- A canonical sequence
  - is a legal sequence that is not equivalent to earlier sequence.
  - represents a unique state of the system.

- The stimuli enumeration process represents a process of creating a Markov chain usage model (MCUM).

- The states of the MCUM are determined by the canonical sequences.

SBS process – Step 6

Step 6. Black box specification – represent canonical sequences using variables



- Each canonical sequence corresponds to a unique MCUM state which is a combination of variable values.
- Defining variables is a manual task.

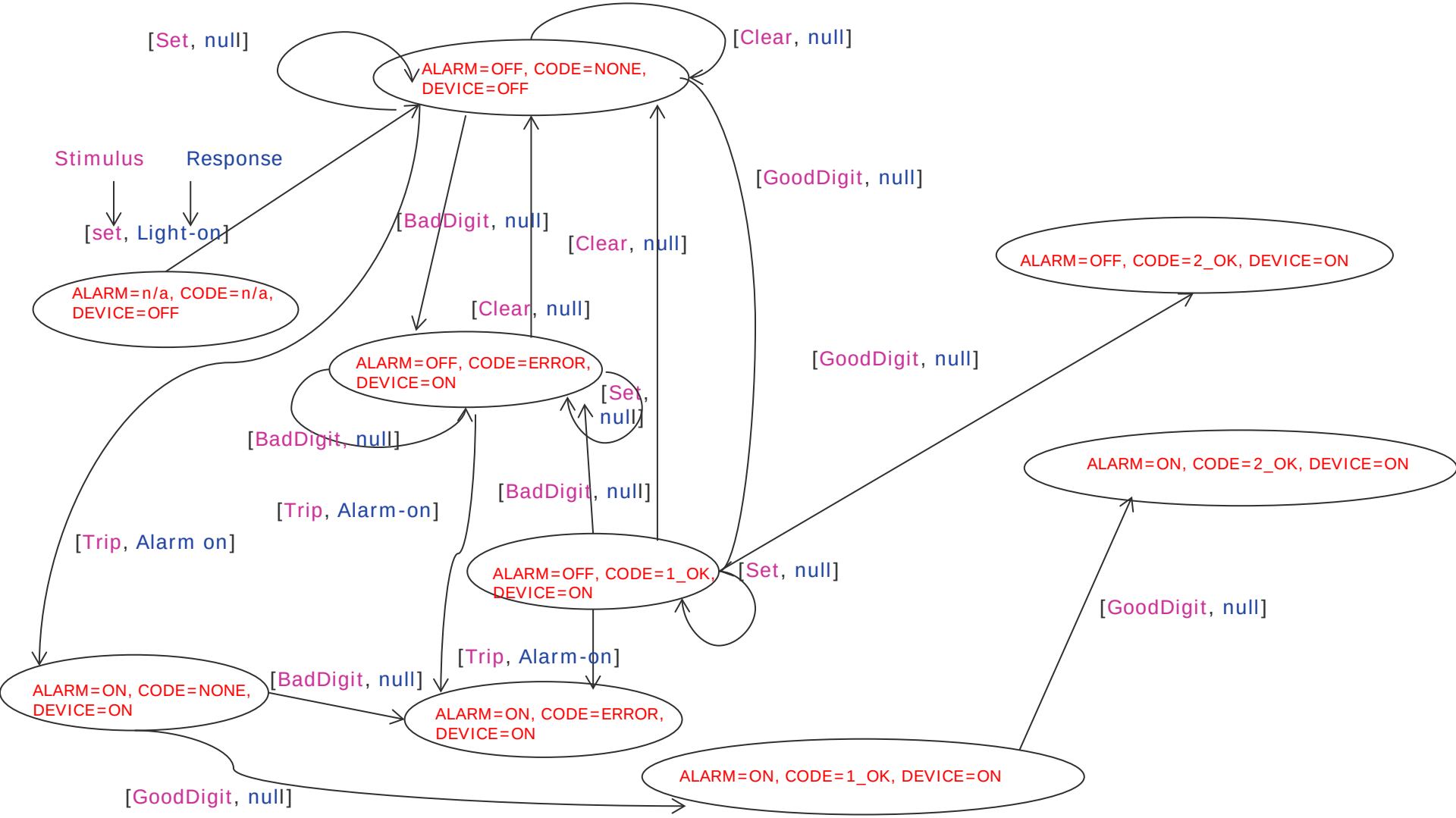
Table 1. Variables and their values

Variable	Possible values
Alarm	“NotApplicable”, “OFF”, “ON”
Code	“NONE”, “NotApplicable”, “ERROR”, “2_OK”, “1_OK”
Device	“NotApplicable”, “OFF”, “ON”

Table 2. Canonical sequences represented by variables

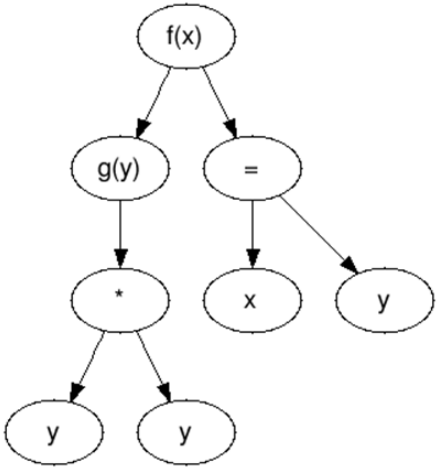
Canonical sequence	Alarm	Code	Device
Lambda	NotApplicable	NotApplicable	OFF
Set	OFF	NONE	ON
Set.BadDigit	OFF	ERROR	ON
Set.GooDigit	OFF	1_OK	ON
Set.Trip	ON	NONE	ON
Set.GoodDigit.GoodDigit	OFF	2_OK	ON
Set.BadDigit.Trip	ON	ERROR	ON
Set.Trip.GoodDigit	ON	1_OK	ON
Set.Trip.GoodDigit.GoodDigit	ON	2_OK	ON

Markov chain model of the alarm software



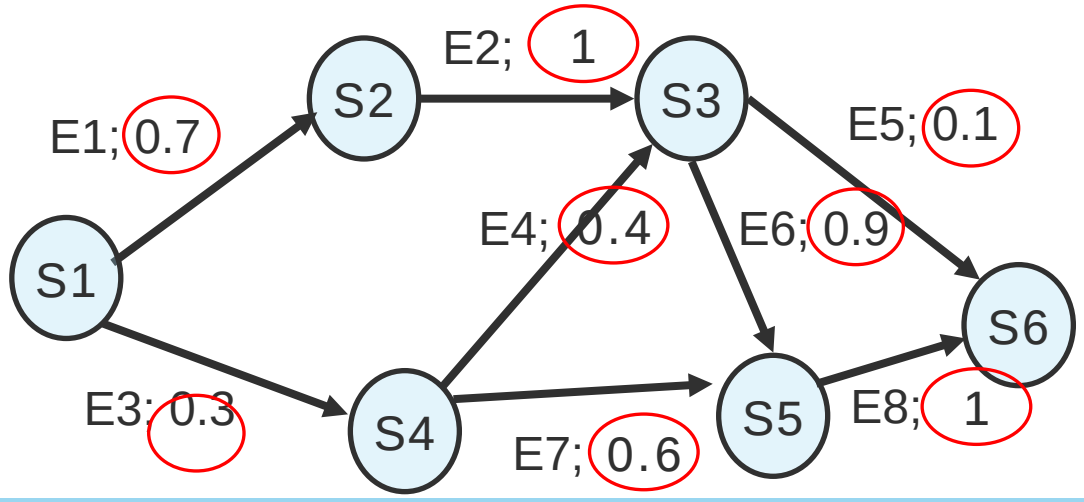
Predicates

- A predicate represent a stimulus that can be expressed in form of a regular *expression rule.*
- *In terms of SBS, predicates are typically used for the following purposes:*
  - *To increase abstraction level, i.e. conditions resulting from specific sequences of detailed stimuli can be modelled as one single predicate having a limited number of possible predicate values, such as true/false*
  - *To define pre-conditions, i.e. the predicates reflect stimuli set outside the border of the model and therefore the predicate value is considered constant.*
- More abstract stimuli in form of predicates can be used to reduce the depth and complexity of the SBS model, and can also be used to decompose the model into sub-models where one or more of the states of the sub-models are predicates in the higher level model.

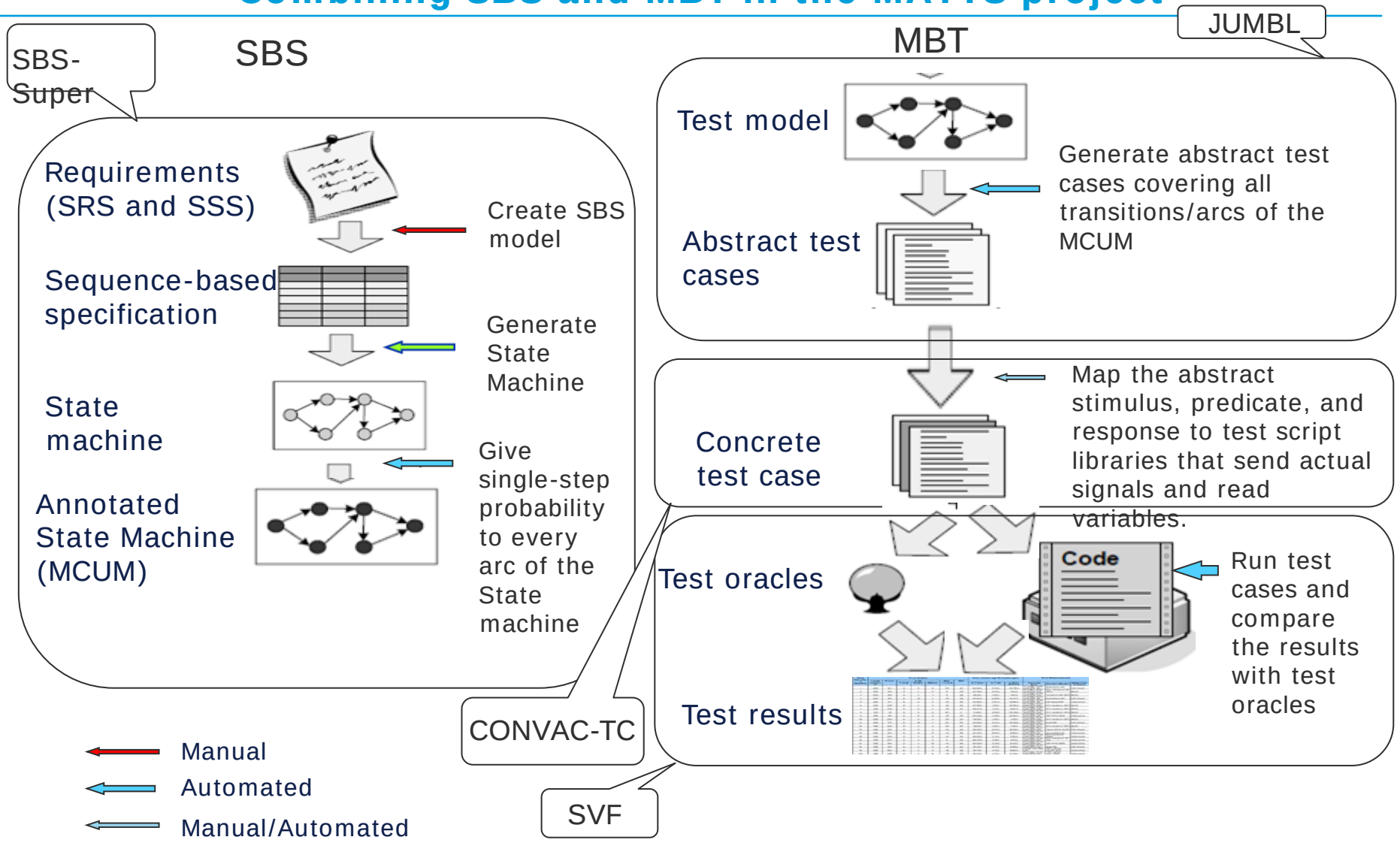


Testing based on SBS outputs

- **Each arc of the MCUM can be annotated with**
  - its single-step transition probability (based on usage or use even probability)
  - how critical the transition is (i.e. weight of the transition)
- **Test cases can be generated based on different strategies**
  - Coverage of each state and each arc (i.e. all state transition rules)
  - Randomly selected based on assigned probabilities/weight
  - Basis path coverage (all combinations of paths only possible if there are no loops and the state machine is not too complex)



Combining SBS and MBT in the MATTS project



Selecting functions for SBS modelling

❑ Scope of SBS modelling

- Central Software of a satellite: V1 & V2

❑ Criteria for selecting candidate functions

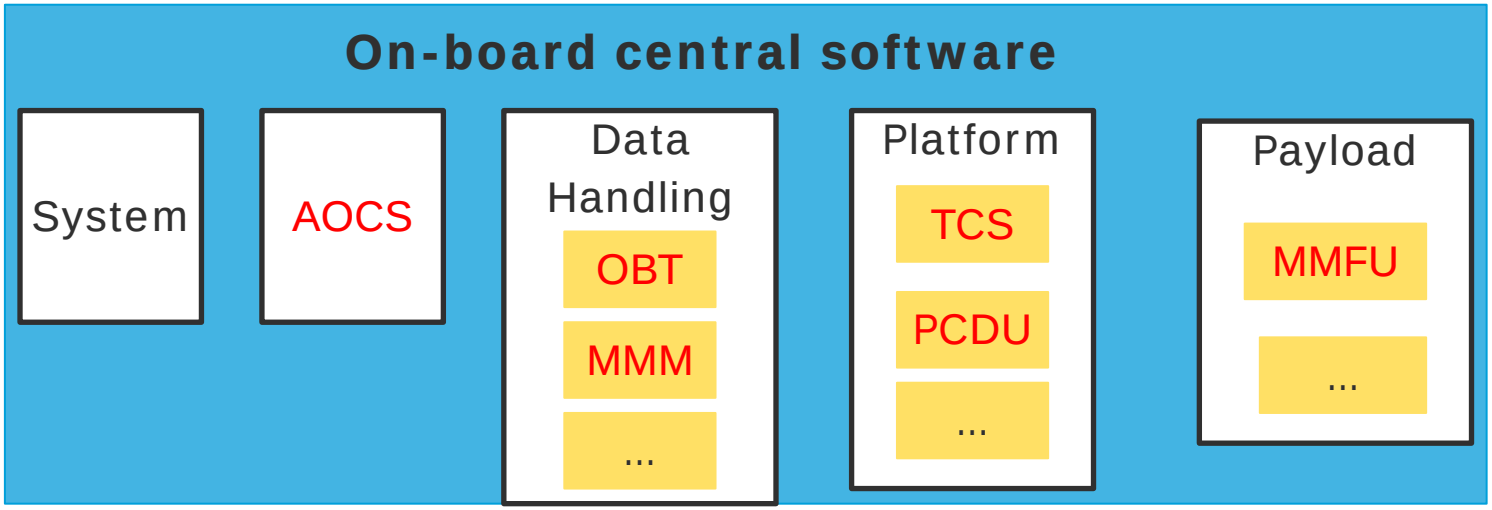
- State machine oriented
- Have a fairly complex combination of input sequences

❑ Requirement classification for SBS modelling

- Cannot be modelled
- Relevant but too simple for SBS
- Relevant and worth modelling

Functions that were modelled

- Thermal Control (TCS)
- Power Control and Distribution Unit (PCDU)
- Attitude and Orbit Control System (AOCS) mode management
- Mass Memory Management (MMM)
- Mass Memory and Formatting Unit (MMFU) management
- On-board Time (OBT) management



Summary of the requirement classification results

	Total	Percentage
Total number of V1 and V2 requirements	1908	100%
Cannot be modelled	753	39%
Relevant but too simple for SBS	346	18%
Relevant and worth doing	809	42%
Selected candidates for modelling, ref. prioritized list	**224**	12%

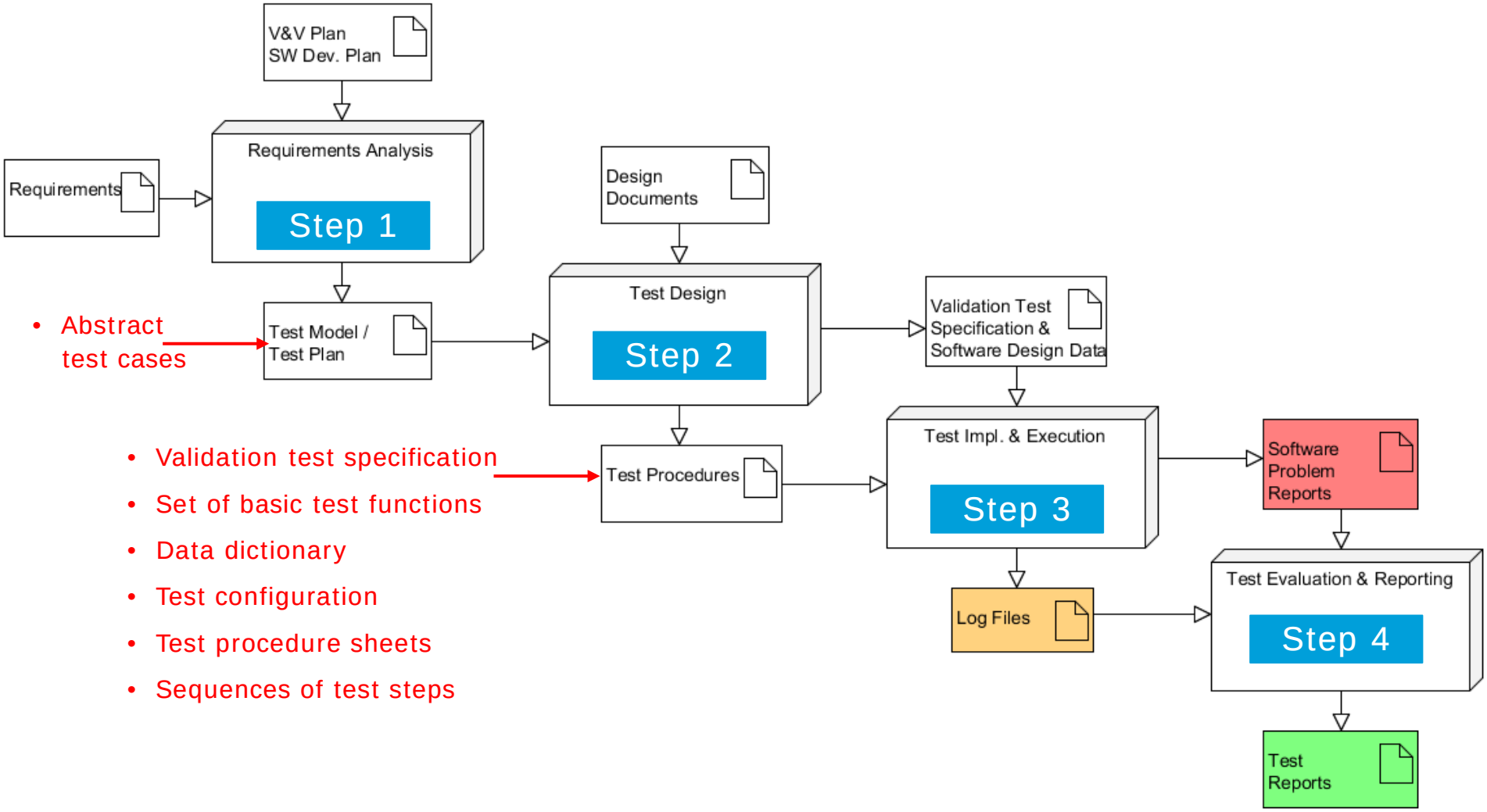
Classification of requirements not suitable for modelling

- Specify TC, TM or PUS Service (29%): Requirements that specify which PUS Services are allocated to the specific functionality and should be instantiated.
- Data management (28%): Requirements related to data management, e.g. requirements specifies data format and the place to store data.
- Implementation (33%): Requirements that specifies the detailed implementations. An example requirement is “The CSW shall call the function X at a rate of 1 Hz to ensure that the packets are written to the packet stores with the correct storage time”.
- Performance (9%): Requirements related to real time performance. An example requirement is “The module X shall process all M control loops within N seconds”.
- Continuous mathematical model (2%): Requirements that are related to responses with continuous values. An example requirement is “the module X shall implement the following algorithm Y”. The algorithm Y is a mathematical model that calculates magnetic torque

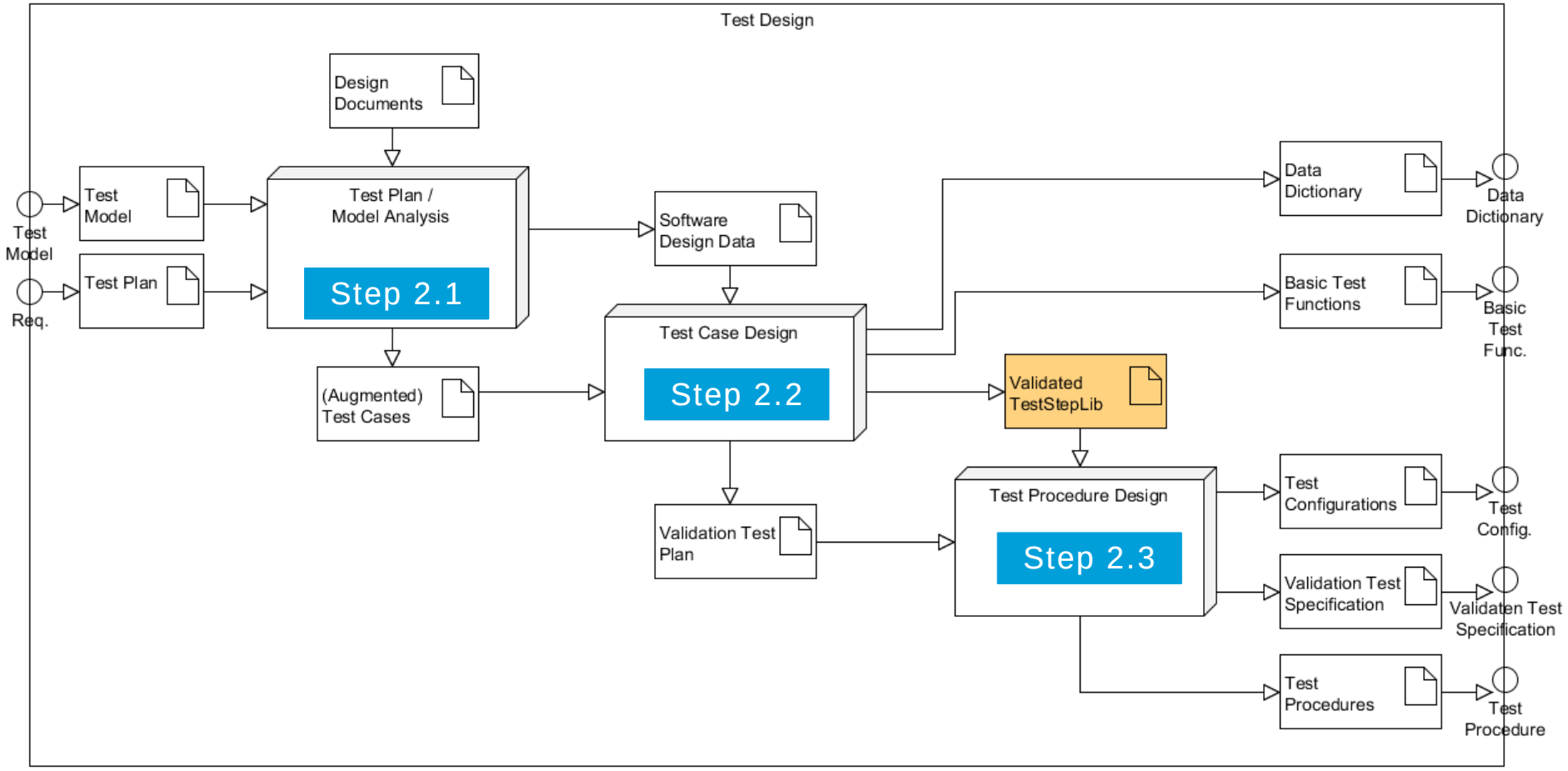
The test case generation tool - JUMBL

- The Java Usage Model Builder Library (JUMBL) have been made by the Software Quality Research Laboratory of the University of Tennessee. http://sqrl.eecs.utk.edu/esp/jumbl.html.
- The Model Language (TML) was used to import the MCMUM from the SBSsuper,
  - No further adaption was needed at JUMBL level.
- A generated test case represent a path of Arcs in the MCUM:
  - For n states, the number of possible state transitions equals n^2
  - Each Arc in the MCUM represent a unique state transition rule/criteria
  - As there may be more than one transition rule/criteria per possible state transition, the number of Arcs may in some cases be significantly larger than n^2
- JUMBL currently supports 4 strategies for test case generation.
  - Generate a set of test cases which visit every arc in an MCUM at least once
  - Generate a set of test cases randomly based on the probability of each arc
  - Generate weighted test cases in order by weight
  - Generate test cases manually in order to meet specific test requirements

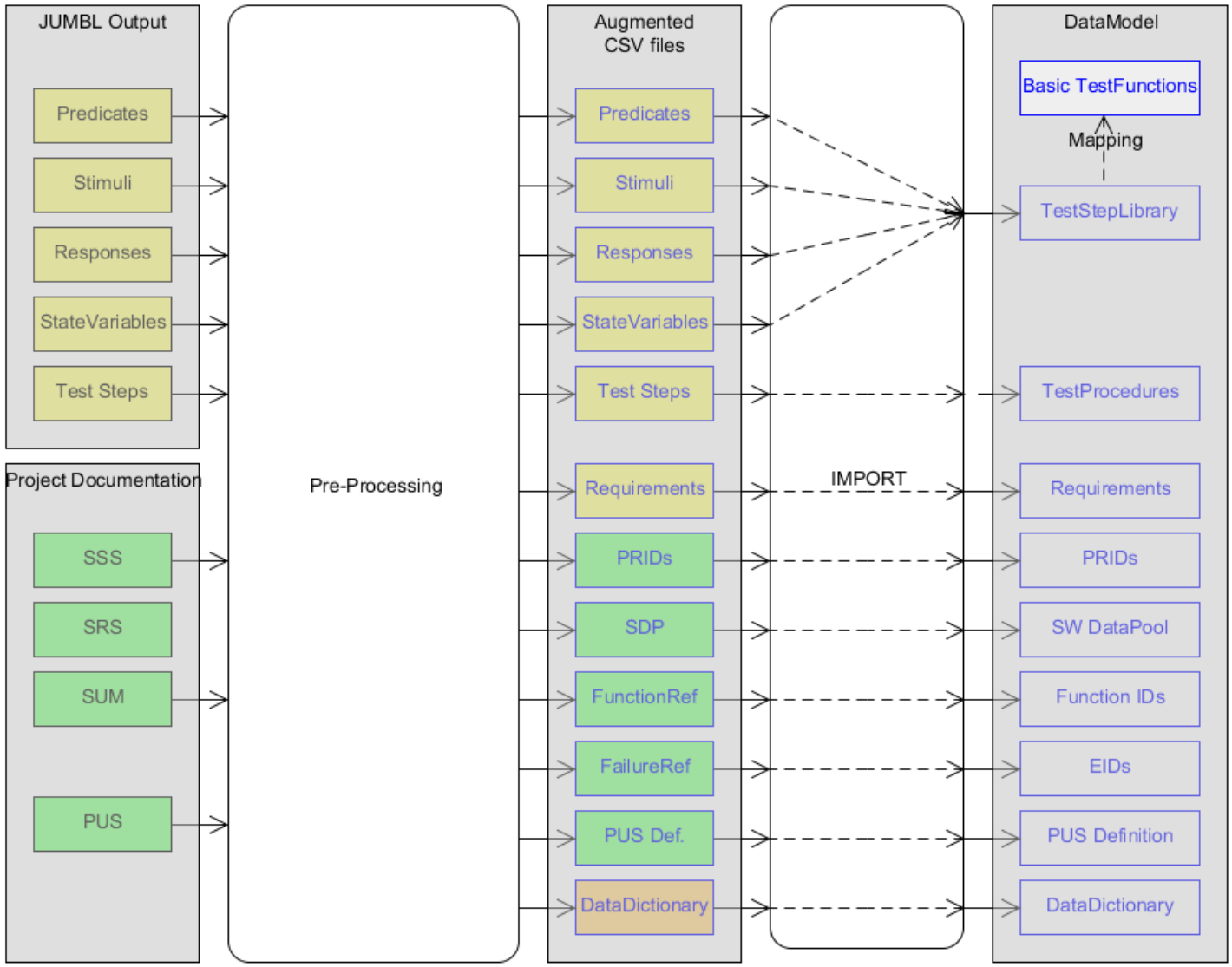
Process for test case definition to test execution and result analysis



Step 2 Test design process



Step 2.1 Map test model files to SPPS data model



Step 2.2 Test case design

- Map abstract test steps derived from the SBS elements (i.e. stimuli, predicates, responses and state variables) to sequences of valid PUS and/or SIF commands
  1. Identify common functions among the abstract test steps
  2. Refine the identified functions by defining the sequence of commands
  3. Assign basic functions to items of the test step library

Examples of basic test functions

Test function	Used in Test Steps
checkAOCSState (mode, submode, sa_status)	21
checkPUSResponse (sst, ecode)	52
getSDP (pid1, val1)	52
setFunctionByID (RID, Param1, Param2, Param3, Param4)	59
checkTCSState (tsw_status, TMTID, TCTLineIndex, T_min, T_max, glb_status)	19

Step 2.3 Test procedure design

- Abstract test steps derived from the SBS models using JUMBL are converted into executable test steps.

- One abstract test step may be converted into several executable steps.
  - E.g.: One abstract test step may contain several responses which need to be checked against test oracles using several executable steps.

Step 3 Test execution

- A Test Procedure consists of the Test Procedure Sheet and an executable file.
- The procedure to run a test is
  1. Select one test from the test tree by clicking on the name
  2. Select the corresponding test procedure and check if all relevant information is provided.
  3. Execute the following activities:
     - Pre conditions: activities to be performed before a test script can be executed.
     - Test execution: activities to start the test and observe the output produced by the test script.
     - Post conditions: activities to be performed when a test script finished execution.

Cost-effectiveness of making SBS models

- In general, 70% of the effort was spent understanding the domain and 30% was spent on filling the data.
- An iterative process to update the SBS model if
  - Adding requirements
  - Deleting requirements
  - Updating requirements

Function	Man-hours spent on making the SBS model	Man-hours spent on updating the SBS model
AOCS Mode management	51	16
Thermal Control	60	16
PCDU	60	28
OBT management	42	35
MMM	86	35
MMFU management	8	10
Sum	**307**	**140**

Time and effort and fault finding effectiveness

Analysis of time & effort	AOCS	Thermal Control	PCDU	OBT	MMFU	MMM
Adapt test script to SVF	26.4	37.4	19.8	15.4	6.6	114.4
Test execution (10h/run)	1.2	1.7	0.9	0.7	0.3	5.2
Result analysis	9.6	13.6	7.2	5.6	2.4	41.6
Cost effectiveness	• Low	Medium	Medium	Medium	Medium	Low
# Requirements model/hr	0.1	0.4	0.4	0.4	0.4	0.2
#Requirements test/hr	0.3	0.9	1.5	1.6	1.5	0.2
#RIDs found	0	3	0	3	2	7

- All RIDs delivered in the modelling phase
- No RIDs delivered in the testing phase (very mature software)

Main challenges faced in the project

- Terms used to represent control system stimuli and responses at requirements level are not the same as the ones used to represent inputs and outputs at code level.
  - Thus a considerable effort had to be used to create a manual mapping between information at the two levels
- Prerequisites in terms of system and software state and a specific configuration of the SVF and the on-board software parameters are required for the respective test cases, as otherwise the tests may fail.
  - Relevant information is often not present in the specification.
  - Thus with the current type of specification, on-board software expert knowledge is required for effective testing of a satellite.

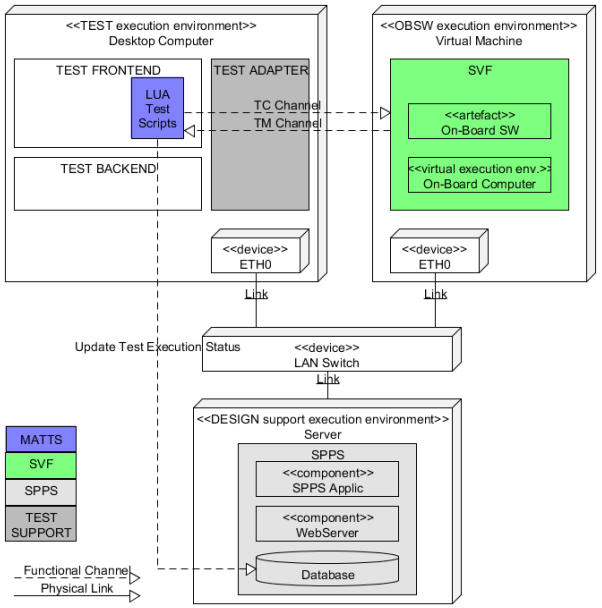
SVF capabilities

In this project Airbus DS has been able to extend an existing test environment, thus the need to develop new test code was relatively limited

In the near future, combining SBS with MBT is considered most relevant for an organisation that can reuse the same SVF in several projects.

- This is mainly because it is necessary to overcome the challenges identified in the previous slide

- Building a test environment for a single project specific SVF to be used for both nominal V&V and ISVV may require more standardisation before it is economically viable.

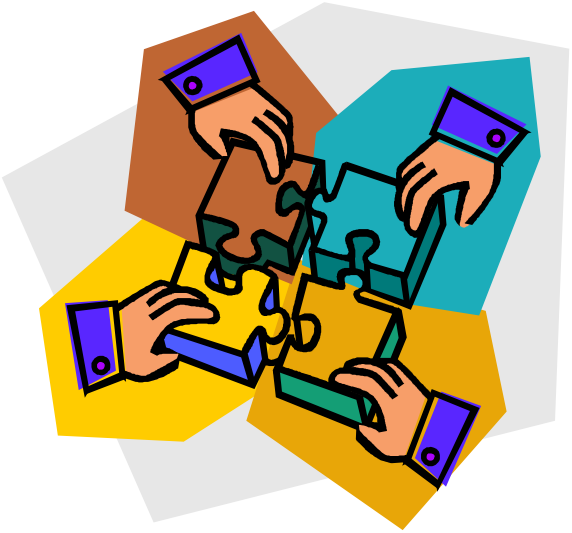


Compliance with ECSS software standards and ESAs guide to ISVV

- SBS can be used for requirements specification, verification and validation both in nominal software development and for ISVV.
  - For those requirements where all aspects are modelled, full requirements test coverage is obtained and traceability can be demonstrated.
- Method must be complemented by manual inspection and also other forms of modelling, testing and analyses since:
  - There are many types of requirements that are not relevant to model with SBS
  - There are a number of V&V related requirements in relevant standards and guidelines that are not covered by this method
    - E.g. using the current tool chain, we will for each state transition rule/criteria just pick one equivalence class value of the relevant stimuli without exercising the possible other values, such as boundary values.

Main benefits

- Improved early validation
  - Formal specifications with clear completeness criteria will reveal omissions in the requirements specifications at an early stage.
- Improved specifications
  - The state machines generated through the SBS methodology may be more detailed, and also different, compared to the ones explicitly expressed in the specifications.
  - State machines may be fed into the development process
- Improved testing
  - tests generated from the SBS model may be more thorough when it comes to testing of state machine oriented functionality, compared to manually developed test
  - No effort spent in selecting and designing test cases



Future improvements

- Potential improvement in SBSsuper tool and Open Source Tool JUMBL
  - Improved support for test of timing related properites
  - Support for exercising all equivalence classes of a given parameter per state transition rule
  - Support for basis path coverage
- Standardisation at specification and SVF level
  - Defining a domain-specific dictionary may avoid part of rework for generating executable test case.
  - Investigate how cost-efficient and unambiguous software requirements should be specified in order to automatically generate executable test cases.

www.dnvgl.com

SAFER, SMARTER, GREENER