A Domain Specific Test Language for Systems Integration

Robin Bussenot
IRIT
118 route de Narbonne
Toulouse, France
robin.bussenot@irit.fr

Hervé Leblanc
IRIT
118 route de Narbonne
Toulouse, France
herve.leblanc@irit.fr

Christian Percebois
IRIT
118 route de Narbonne
Toulouse, France
christian.percebois@irit.fr

ABSTRACT

In avionic context, systems are complex, embedded, critical, reactive and real time. In this context, testing activities are predominant in a V development process. We propose to bring in some features coming from agile methods. System integration testing that means systems are tested individually and together in order to ensure that they all operate correctly. We focus on functional and system integration testing just before the ground testing phase. Nowadays, test procedures and test plans we studied are described in textual manner and are executed manually. We aim to provide a common specific language that improves communications for the team of test designers and between test designers and test performers. In the same manner as agile test frameworks, this language allows to structure the test procedures. Moreover, this language tends to facilitate the automatic execution of some parts of the procedures. We choose a domain specific language approach to design a first domain specific test language dedicated to networks system integration.

CCS Concepts

•**Software and its engineering** → **Domain specific languages;** *Software testing and debugging;* •**Hardware** → *Board- and system-level test;*

Keywords

Test system; Test procedure; Domain Specific Language;

1. INTRODUCTION

In avionic context, systems are complex, embedded, critical, reactive and real time. These characteristics demand to increase the effort needed to test the behavior of all systems that composed an aircraft and also an effort at the requirements phase. Moreover, the whole development process in this context follows a classic V-model. Testing is principally done in the integration phase (right side of the V-model).

In system context, a test bench is needed to interact with the System Under Test (SUT). These test benches can realize automatic or manual tests with human intervention on the bench to execute them step-by-step. Our work was supported by a FUI[1] project named ACOVAS aiming at introducing agile methods around a new generation of test benches. The initial observation is that the verification and validation process is the Achilles' heel of complex systems development. In this project, our contribution would consist in providing a support allowing Test First design. However, a necessary condition to adopt this approach is to be able to produce unit tests. Then, we propose a first Domain Specific Test Language (DSTL) dedicated to a specific field for system integration testing in avionic context.

We are working on system integration testing phase. Testing in this context is supported by test procedures written in natural language in a textual form. The execution of those tests remains mostly manual. Test procedures are pulled from design office requests which gives as input test plans with their related objectives. As several people are implied in testing activities, test procedures are a very important way of communicating. But the use of natural language could produce several errors due to misunderstanding of testers who design and execute the test procedure. Producing a Domain Specific Language (DSL) for them will improve communications between testers and avoid misunderstandings, misinterpretations, unsaid things, ambiguities that can arise with natural languages. And finally, producing a formal language dedicated to test procedures is a precondition to generate executable scripts on a test bench.

To better understand our contribution, we introduce some basic concepts as testing in avionic system and DSLs in section 2. To introduce our DSTL, we present test procedures written in natural language and their use of these in the work flow document process in section 3. Our contribution is detailed in section 4. The structure of our DSTL divide flight plan test procedures into unit tests. We classify elementary instructions into five categories (Step, Trace, Check, Log, Reminder).

In section 5, we compare our work with other dedicated test languages, other approaches to disseminate agile methods in embedded system development and with scenario based languages. We summarize our contribution and give future developments of our work in section 6.

XP '16 Workshops, May 24 2016, Edinburgh, Scotland Uk

DOI: http://dx.doi.org/10.1145/2962695.2962711

[1]Fonds Unique Interministériel (FUI) is a French program dedicated to support applied research, to help the development of new products and services susceptible to be marketed in court or middle term.

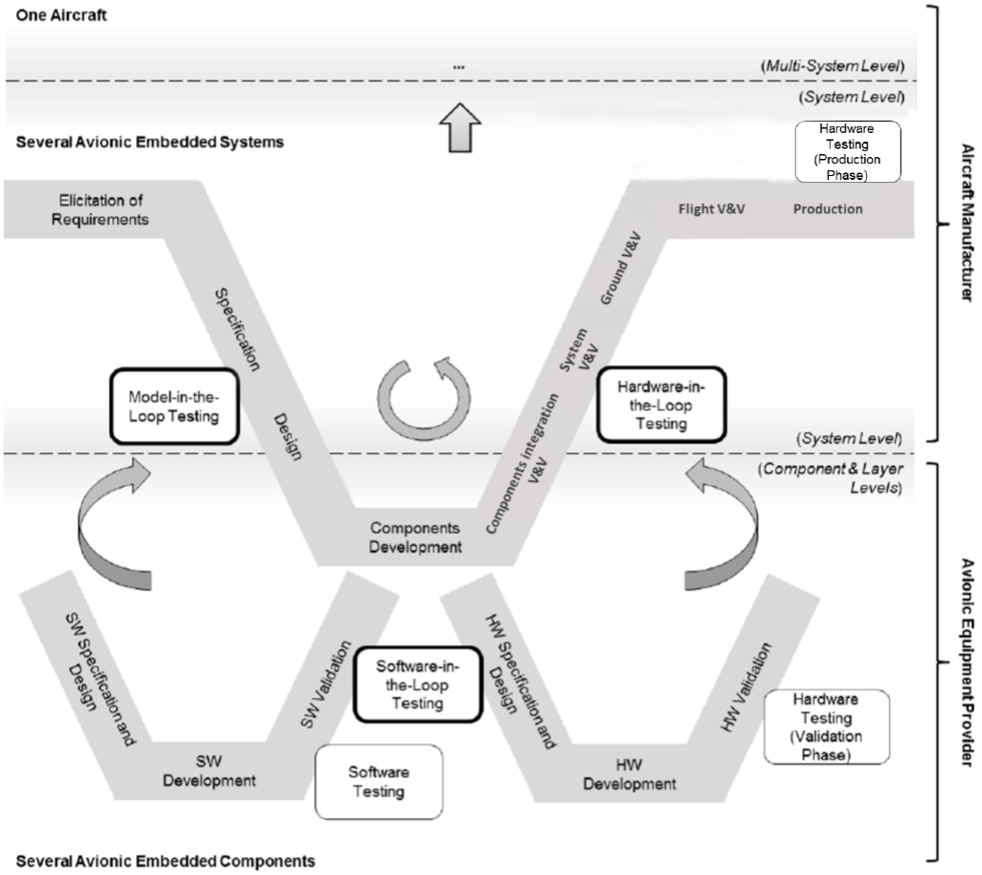


Figure 1: The W-model and testing activities.

2. BACKGROUND

In this section, we present concepts necessary to apprehend our work: testing avionic systems with their W-model, details about Domain Specific Languages (DSL) and integrated environment development for the design of new computer languages.

2.1 Testing avionic systems

After presenting the W-model of testing activities, we focus on Hardware-in-the-Loop testing during the system integration testing phase.

2.1.1 The W-model and testing activities

Tests in avionic context act for a validation and a verification process. Validation is the process of making sure that the final product will reflect customer expectations. Verification is the process of justifying that the system follows its specified requirements. Verification and validation will be performed at both requirement and implementation levels [14]. System is complex, system of systems is more complex and then an important part of the validation and verification activities is performed by tests.

An airplane is composed by many complex and embedded systems and the process of development of each system follows a V-model. Figure 1 presents a V-model for one of them. The bottom part of the Figure 1 shows two V-models that occur in parallel. These two V-models are named the W-model and generally the system resulting from this process is done by an avionic equipment provider. The V-model on the left side corresponds to the software process development and the V-model on the right side corresponds to the hardware process development. The result of these two processes is integrated in a main process at the system level that follows a V-model too. System and multi-system levels are the fields of an aircraft manufacturer.

There are two kinds of testing activities: classic test in isolation and In-the-Loop Testing (surround with bold box). In-the-Loop Testing provides models and simulators to put the System Under Test (SUT) in simulated flight conditions. Simulation of the environment stimulates the SUT with input data in order to produce output data to be analyzed.

This kind of testing was implemented to allow integration testing as soon as possible. The environment model concerns collaborations between the SUT and other systems. A real time simulator embedded in the test bench executes models to simulate test context environment. This is needed to be able to test integration of components before getting all real systems available. The main In-the-Loop testing activities are: Model-in-the-Loop Testing (MiL) , Software-in-the-Loop Testing (SiL) and Hardware-in-the-Loop Testing (HiL). Classic testing activities are Software Testing and Hardware Testing.

Testing is also performed since requirements level (MiL). Even if at this moment there is no physical equipments nor

software to test, a simulation model is executed to ensure that requirements are correct. In SiL testing, the real software is considered and tested with an emulated hardware. In hardware testing, the real hardware is considered to ensure compatibility with its operating system. In HiL testing, the real software is integrated into the target hardware.

Our proposition of DSTL matches with HiL testing activity. From this activity in the V-process, the denomination hardware will refer to the real software with the real hardware. They are tested together by the HiL testing activity. HiL testing is the main activity of the system integration testing phase we detail below.

2.1.2 System integration testing

The class of problems managed corresponds to shared resources for avionic modules and network testing. There are three kinds of testing: functional testing, structural testing and random testing. Functional testing correspond to a black-box approach and structural to a white-box approach. System integration testing (named System V&V in the figure) corresponds to functional testing. System integration testing and multi-system integration testing mean that systems are tested first in isolation and then together in order to ensure they all operate correctly.

The execution of a test will be manual if there is no formal test specifications or if the test bench does not support all the automation. In any case, test execution can be automated or semi-automated if test specifications are encoded in a computer language [14].

In our case, integration testing corresponds to network systems and tests are principally manual. Testers will provide inputs to the SUT directly managing the interface provided by the test bench, for example: pressing button, cutting cable, power off a component .... Depending on the domain and due to the complexity of the SUT, some acceptance criteria cannot be formalized and even if it is possible, test oracles cannot be designed or implemented. Testers must manage tools manually to collect data that will be analyzed later to give a verdict for some tests.

2.2 DSL, DSTL and languages workbenches

To present the result of our work, we recall the notion of DSL that we have extended to DSTL (Domain Specific Test Language). To design our DSTL, we have chosen a language workbench. Then we present language workbenches, which are rising IDE specialized to produce small specific languages [5].

2.2.1 Domain Specific Language

A DSL is a computer language that allows to provide a solution for a particular class of problems [6]. We can oppose it to a general purpose language (GPL) as Java or PHP that can respond to any class of problems. We can cite for examples SQL, HTML, scripting languages that permit building applications, XML configuration languages ...

Producing a DSL for a specific domain brings many advantages [22]:

- a DSL makes easier expressing domain concerns,
- a DSL hides GPL complexity,
- a DSL is used by domain experts that not necessarily have computer knowledges,
- a DSL can generate many lines of code in GPL from few lines,
- a DSL improves team communication and increases the common knowledge about a domain,
- a DSL is managed by specific tools as an IDE for GPL languages (Eclipse for example).

DSLs can be executed following two patterns: external and internal [6]. Internal DSLs extend the syntax of a host language. The most popular host language we can cite is Lisp. External DSLs bring their own syntax and produce executable code in a GPL language.

Our language is an external DSL focusing on the description of test scenarios in an avionic context. This language is dedicated to test engineers only. In fact, our language addresses the problem of writing test scenarios for a particular business domain. We named Domain Specific Test Language this new category of DSL.

Since this kind of language is dedicated to test, it has to follow the AAA pattern (Actor-Action-Assertion) that comes from software engineering. We can quote some existing DSTL like TDL [4] (Test Description Language) aiming at specifying test descriptions.

2.2.2 Language Workbenches

Language workbenches are tools to manage and help the implementation of new languages [5]. We can list most popular of them: Xtext [8], Spoofax [21] and Meta-Programming System [9]. All these tools are doing almost the same thing, but with a different vision and a different approach. The Table 1 summarizes the vision and the approach for the most popular language workbenches.

Table 1: Projectional vs Parsing

	CS first	AS first
Parsing	Xtext Spoofax	Xtext
Projectional		MPS

The vision concerns program editors for new languages. Editors can be projectionals as done in MPS for example, or textuals as usual in Xtext and Spoofax for example. In projectional editing, the abstract representation is the core definition of the system. Then, the language workbench manipulates directly the abstract representation and projects an editable representation for the programmer. According to Martin Folwer in [6]: "Projectional editing thus usually displays a wider range of editing environments - including graphical and tabular structures - rather than just a textual form." The separation between concrete syntax and Abstract Syntax Tree (AST) permits a better control than a unique textual editor.

Two ways to design a DSL exist, meta-model first approach (Abstract Syntax first) or Concrete Syntax first approach. The meta-model approach is issued from MDA (Model Driven Architecture) community and matches with a top-down approach, while the concrete syntax approach focuses on the syntax of the language and then is considered as a bottom-up approach.

To choose the language workbench between Xtext and MPS, we first implement two prototypes of a test language resuming JBehave principals. We applied these principals

to the multiplication class example issued from the tutorial of JBehave. Scenarios are expressed in a Gherkin language and then are derived into executable JUnit tests.

Table 2: JBehave in Parsing vs Projectional

Xtext	MPS
Grammar in EBNF	AST concepts
	Editor for each concept
Generation with template in textual manner	Generation with template in projectional editor

The Table 2 presents the steps to implement the prototype. With MPS, the programmer is guided by projectional editors corresponding to each concept of the AST. With Xtext, the programmer has an empty text frame and must know the grammar of the language to produce test scenarios. With MPS, no syntax errors are allowed. With Xtext, syntactical errors are allowed and we must manage error messages. Xtext provides a small stub generator with a method `doGenerate()` and a set of primitives to address concrete syntax trees. MPS provides a model-to-model transformation approach [1] by linking each concept of the JBehave AST with concepts of the Java AST. Code templates in MPS are coded with a projectional editor that prevents syntax errors.

For those reasons, we have chosen MPS. Moreover, MPS permits to add constraints to each concept helping programmers to avoid other semantic errors and displays errors on the incorrect code directly. It also, provides features for language composition and language extension [2].

3. CONTEXT

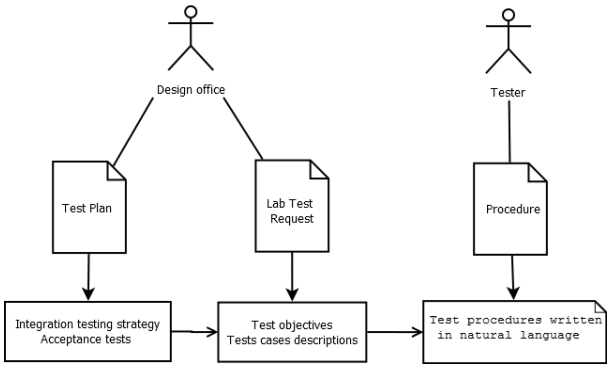
We have set of 20 test procedures dedicated to network system integration at our disposal. First, we explain the workflow of documents in the specification process of test procedures. The second part presents test procedure documents. Then, we finish by explications about some instructions.

3.1 Workflow of test procedures

In this process, design office and testers are the two stakeholders. The workflow of documents is illustrated in the Figure 2.

First the design office produces a test plan providing integration testing strategy for a specific system, and acceptance criteria defined from system requirements. Then, the design office derives test plans into Lab Test Requests (LTR). A LTR is specific to a campaign of tests while a test plan is more generic. A campaign of test is related to a version of the SUT. A LTR is composed by test objectives. A test objective is described by a single sentence and is completed by test conditions on the SUT from which results of a test can be exploited, a description of a scenario independent of the test means, and expected results. Finally, LTRs are linked into concrete test procedures by a tester. A test procedure contains a description in a graphical language of test means and a sequence of test chapters. Test means consist in test bench architecture around the SUT and logical links between the test bench and the SUT represented by ICDs (Interface Control Document). Test chapters represent executable scenarios on a test bench based on the descriptions owned by the LTR. For each concrete scenario corresponds a test block that contains instructions described in natural language.

Figure 2: Test procedures workflow



3.2 Test procedure documents

We got as input for our experimentation 20 test procedures. A half contains procedures with tagged instructions. Considering only procedures with tagged instructions brings us a pre-classification which facilitates the study. Overall, the corpus studied is composed by 3708 instructions contained in 10 procedures. A document procedure is composed on average by 360 instructions, but it could vary from 70 to 1700. The biggest document comes with 118 pages of word text containing 17 chapters split into many test blocks (about 110 test blocks). A test block corresponds to the execution of one test.

Nowadays, these test procedures are executed manually on a test bench. Instructions are written in natural language and would describe atomic actions that a tester performs on a test bench. Integration testing must be performed on several sites. That is why, test procedures must follow the same template and the natural language used has to be simple in order to reduce ambiguities. The existence of a first classification of instructions demonstrates the effort of testers team to speak the same language.

However, instructions remain in natural language that does not permit automation of tests, does not assure unambiguous instructions, does not allow any code duplication and does not restrict the number of ways to express the same idea.

3.3 Instructions

Procedures expressed by testers are composed of five kinds of instructions:

Step: an action is performed on the SUT, or on the test bench, or on a tool.

Check: a verification is performed about the state of the SUT. There are two contexts of use: a precondition on the SUT to perform the test and a verdict on the test.

Log: an action to store some parameter values involved in the test.

Trace: an action on a tool in order to record relevant test data.

Reminder: instruction addressed to a tester concerning the management of test executions.

Table 3: Number of instructions by type

	Step	Check	Log	Trace	Reminder
Number	1274	1047	594	520	269
%	34,4	28,3	16	14	7,2

The distribution of these kinds of instructions is given in the Table 3. Although we are in a hardware context, test procedures follow the AAA pattern (Actor-Action-Assertion), a majority of instructions concerns actions (tagged by step) and assertions (tagged by check). At least, 30% of instructions manage test results data in order to be analyzed later. The goal of Trace instructions is to start and stop tools needed to log outputs or to spy system behavior. The goal of Log instructions is to get some specific variables or parameters to store them.

Unfortunately, even if test procedures are tagged, some inconsistencies occur. For example, we can have instructions not correctly tagged:

```
[STEP]Wait for a SID, in order to check the MODULE config-
uration.
[/STEP]
```

The semantic of this instruction is when SID signal is received a check is performed to test the configuration. Then the type of this instruction is a check, more precisely a precondition of the test.

For another example, an instruction can be composed by many sentences that could be classified in a different kind of instructions:

```
[STEP]Switch on all others MODULES of the same cluster,
with full configuration of loads.
Then link all the modules of the cluster to the AFDX net-
work.
With Tool1 or Tool2 check that all SB of these MODULES send
failure
messages both with IMA bit send to 1 and 0.[/STEP]
```

First and second sentences are two step instructions while the last one is a check instruction. Seeing the complexity of test procedures, even if they are tagged, the goal of our DSTL is to improve the way to produce test procedures.

4. CONTRIBUTION

The language we provide is a DSL (Domain Specific Language) [22], more precisely a DSTL (Domain Specific Test Language). To design this language, we follow a bottom-up approach. After studying test procedures from network avionic context and by analogy with xUnit test frameworks we propose a structure for this language. The structure of the DSTL consists of all concepts necessary to contain test instructions. We are working on test instructions with a natural language processing [12] approach and we give results for instructions tagged by trace.

4.1 From Word to xUnit tests

We change and complete the structure of Word documents to match with concepts from xUnit frameworks. In xUnit frameworks, all concepts for unit testing match with the Actor-Action-Assertion pattern. Moreover, frameworks of unit tests provide some mechanisms for refactoring code of the tests (setUp() and tearDown() methods). Using concepts of unit test into integration tests improves readability and value of these tests. Integration tests will not be used for verification and validation purposes only.

The Table 4 gives a correspondence between a test procedure in our context and a test suite in xUnit frameworks. In software engineering, an actor corresponds a class instance class to be tested but in our context it corresponds to the SUT connected to the test bench. The instructions which denote an action correspond to instructions tagged by Step and Check. Sometimes, some preconditions occur on the state of the SUT before starting the check of the test. A test can give a verdict if and only if its precondition succeeds. The instructions which denote an assertion correspond to instructions tagged by Check and Trace. In avionic system context, there are a lot of test objectives that are not specifiable in the form of check instructions. Then, testers use many tools to record tests data to be analyzed later with a data viewer in order to provide a verdict for the test.

The tear down concept is not transposed in our language because we applied a bottom-up approach to a corpus of manual test procedures where the tear down action is implicitly done by the tester between each test. However, in an automatic execution context, we think that a tear down concept could be useful.

Table 4: Mapping between xUnit concepts and system integration testing concepts

xUnit concepts	**Integration test concepts**
AllTests	Test procedure
TestCase	Test chapter
setUp()	Set up
tearDown()	?
test()	Test
Actor	System Under Test
Actions	Step
Assertions	Check,Trace
Green bar	all tests are OK
Red bar	at least one test failed
Orange bar	at least one precondition failed

In software engineering, unit testing aims to isolate test concerns and the execution order of unit tests in a test suite can be variable. But in a system context, testers play to put the system into an error state, in order to ensure that the reaction of the SUT is conform to the requirements. This kind of systems is called reactive. For example, when a test must show that an alarm is raised when a threshold is broken (first test), the system has to return in a normal execution state by itself (second test). In this case, we authorize the tester to do these two tests in a unique integration test. Moreover, tests usually follow a nominal flight plan to ensure a correct behavior of the SUT in the nearest conditions of a real flight. Then, the test suite must follow the flight plan and the execution time is expensive. So we cannot constrain testers to isolate each of integration test as the same manner as software engineers. However, each integration test in avionic context must concern one and only one test objective in order to decrease time debugging for the SUT or for the test itself.

We formalized the setup mechanism that was already existing in Word test procedure documents. The Figure 3 shows the setup block that will be executed first for each test block. A setup block is not dedicated to the configuration of the test bench or tools around the bench, but it has to handle the initialization of the SUT to put it in the expected state. This corresponds to the ”given” clause of

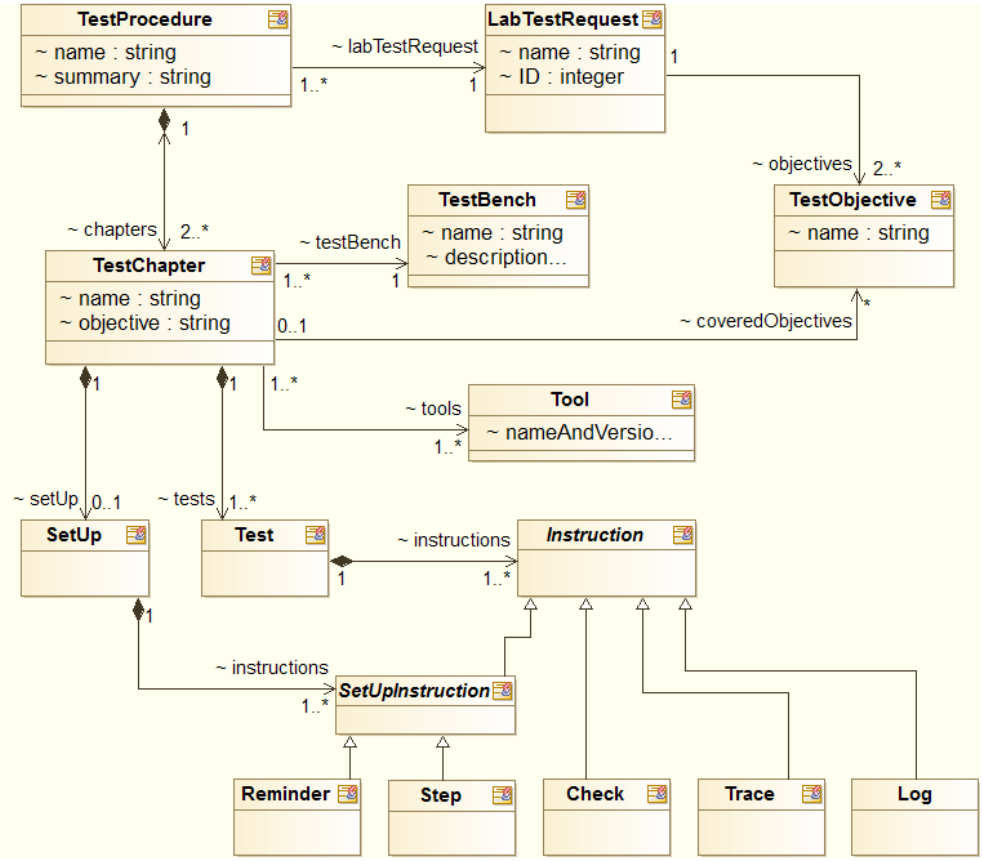


Figure 4: Structure of the DSTL

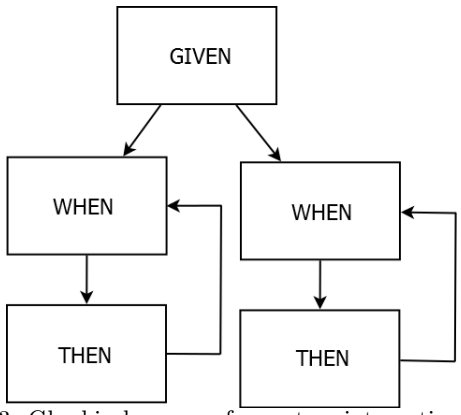


Figure 3: Gherkin language for system integration testing

a Gherkin language [18]. Unfortunately, we can have several "when-then" clauses in the same unit test that concern the same test objective. One of the main purpose in using our DSTL is to minimize "when-then" clauses redundancy in order to reach the best practices of unit tests.

4.2 Language structure

The structure of our language is presented in a meta-model form in Figure 4. Each concept of the AST is related to a class in the meta-model. In accordance to the metaphor with xUnit (see Table 4) and the workflow of test procedure documents (see Figure 2) we have defined concepts in our language.

The main purpose of our DSTL is TestProcedure. A TestProcedure refers to one LabTestRequest that owns TestObjectives. A TestProcedure is composed by several TestChapters and at least two are needed to produce a consistent TestProcedure. To place the TestProcedure correctly in the workflow document, we have added additional concepts as enumeration types: LabTestRequest, Tool, TestBench and TestObjective. TestObjectives are shared by LabRestRequest and TestChapter in order to generate a traceability matrix that ensures the cover of TestObjectives by TestProcedures.

The container of instructions are classes SetUp and Test. In Words document procedures, a file procedure can reach over hundred pages due to the high number of test chapters. With our language, we supply to the user an interface for the definition of test blocks separated from the interface supplied for the procedure. Now a test chapter just contains a reference list towards test chapters: a simple Ctrl+Click on a reference of a chapter will bring us to the corresponding chapter file.

4.2.1 *Formalization using OCL constraints*

To complete the formalization, we give some additional constraints in OCL language. First, we put uniqueness constraints on some concepts in our language.

```
context TestProcedure
inv:TestProcedure::allInstances.name->isUnique(n|n)
context LabTestRequest
inv:LabTestRequest::allInstances.name->isUnique(n|n)
```

Figure 5: MPS concept constraint for covered objectives

```
final nlist<LabTestRequest> allLTR = model.nodes(LabTestRequest);
final nlist<TestProcedure> testProcedures = model.nodes(TestProcedure);
final node<TestChapter> currentTestChapter = contextNode.ancestor<concept = TestChapter, +>;
final sequence<node<TestProcedure>> procedure =
    testProcedures.ChapterRef.where({~it => it.IDTestChapter.id == currentTestChapter.id; }).
    select({~itProc => itProc.ancestor<concept = TestProcedure>; });
final node<CoveredObjectivesRef> me = exists ? contextNode as CoveredObjectivesRef : null;
final sequence<node<TestObjective>> alreadyRefButme
    = currentTestChapter.CoveredObjectiveLTR.where({~it => it :ne: me; }).select({~it => it.refObjective; });
final sequence<node<LabTestRequest>> currentLTR =
    allLTR.where({~itLTR => procedure.LTRRef.all({~it => it.labTestRequestRef.id == itLTR.id; }); });
sequence<node<TestObjective>> autocomplete =
    currentLTR.objective.where({~it => alreadyRefButme.all({~idOK => idOK :ne: it; }); });
return new ListScope(autocomplete) {
  public string getName(node<> child) {
    child : TestObjective.name;
  }
```

```
context TestObjective
inv:TestObjective::allInstances.name->isUnique(n|n)
context TestChapter
inv:TestChapter::allInstances.name->isUnique(n|n)
context TestBench
inv:TestBench::allInstances.name->isUnique(n|n)
context Tool
inv:Tool::allInstances.nameAndVersion->isUnique(n|n)
```

A test chapter must cover consistent TestObjectives. A Covered objective by a test chapter has to belong to the LabTestRequest referenced in the procedure which contains this test chapter.

```
context TestChapter
inv:self.ancestor.labTesRequest.objectives->
includesAll(self.coveredObjectives)
```

At block instruction level, we added two constraints. We must also ensure that a TestChapter contains at least one Step instruction.

```
inv:self.setUp.instructions->
union(self.tests.instructions)
->exists(su|su.oclIsKindOf(Step))
```

We must also ensure that a test contains at least one Check or Trace instruction.

```
context Test
inv:self.instructions->
exists(su|su.oclIsKindOf(Check)
or su.oclIsKindOf(Trace))
```

We do not allow a tester doing check or trace in a setup block because its goal is to put the SUT in test conditions and not to check or trace anything.

4.2.2 *Formalization in MPS*

All constraints owned by the meta-model and described in OCL are implemented in MPS.

In order to help the comprehension of the concept constraint of the Figure 5, we recall some definitions about constraints in MPS. The abstract representation of the language is given on the form of a hierarchy of concepts. A concept can extend another concept, and two concepts can be linked by composition or by aggregation. A Concept defines the structure of a node of the future AST representing code written using a programming language. An abstract syntax tree (AST) is a tree representation of the abstract syntactic structure of source code written in a programming language. Each node of the tree denotes a construct occurring in the source code.

Constraints in MPS are designed for two purposes. Type system constraints allow to report semantic errors to the end user of the language, for example a constraint that ensures that the name of a test procedure is unique. Concept constraints allow to prevent semantic errors by guiding the end user of the language. For example, the constraint of test objectives covered by a test chapter is implemented by a concept constraint. Each constraint has as a starting node representing a concept. From a concept, MPS can give all instances of this concept present in the project, and MPS permits to navigate through relationships between concepts.

The constraint of the Figure 5 is applied when a test designer wants to link test objectives to a test chapter. The designer is writing a test chapter and would add a new test objective. The starting node of this constraint is a new reference of a test objective in a test chapter. This constraint is implemented by a code that must return a sequence of test objective nodes (ListScope in the code) that populates the auto-completion list in the code editor. A test chapter is owned by a test procedure. This list will be only composed by test objectives provided by the Lab Test Request referenced in the test procedure except test objectives already referenced in the test chapter.

More precisely in the code, the variable *procedure* will contain the test procedure owning the current test chapter. The variable *allLTR* contains all test objectives present in the project. The variable *currentLTR* contains all test objectives referenced in the procedure by a process of filtering on the variable *allLTR*. And finally the variable *autocomplete* contains only remaining test objectives by the same process. This variable will be return into a *ListScope* that provides an auto-completion list in the MPS editor.

This constraint ensures that the end user cannot do mistake due to the use of the auto-completion list.

4.3 The case of Trace instruction

We are working on test instructions and we present the state of our reflection on the kind of instructions that we named Trace. The Table 5 provides a global vision of which type of action is often used.

A first manual analyze of the corpus of test procedures gave us a list of sentence patterns. A pattern is a sentence starting with a verb in an imperative form. We recall that a procedure is a list of instructions performed by a tester.

Table 5: Semantic classes of Trace instructions

Semantic class	Verb	Number of instructions	%
Start and Stop	Start	193	37,1
	Stop	171	32,9
			70
Collect	Make/Take screenshot	64	12,3
	Save	12	2,3
	Collect	12	2,3
	Log	11	2,1
			18,7
Perform	Perform	32	6,2
	Compute	22	4,2
			10,4
Total		517	99,4

We classify patterns of Trace instructions in three semantic classes: Start and Stop, Collect and Perform. This classification is manual and involves knowledge about the semantic goal of the action carried by the verb.

With StanfordCoreNLP [12], a Natural Language Processing toolkit, all sentence instructions are transformed in syntax trees with the parser feature. Then, we used requests syntax trees in order to classify verbs into semantic classes. Main instructions are dedicated to start and stop a trace tool (70% of instructions tagged by Trace). Others instructions are dedicated to collect and transform complementary information not present in a test result analysis file.

Start and Stop.

Requests on the NLP tool permit us to notice that Start and Stop instructions share the same structure: a verbal group followed by a nominal group as seen in the Figure 6. Then we added two instructions in our language: `Start [Tool*]` and `Stop [Tool*]`.

Collect.

For now, we take into consideration the action take screenshot with the instruction `Take a screenshot of [Tool] GUI`. We asked the testers team about a unique verb to replace occurrences of save, collect, and log. They choose the keyword collect to save a report from a tool. The syntax of this instruction is : `Collect [Tool] report`. However, instructions tagged by Log denote a list of variables that must be saved in a test result analysis file. The log verb in the context of instructions tagged by trace means that saved elements are stored in complementary files.

Perform.

Sometimes, a processing on test data is needed before they are stored. The Perform semantic class represents this kind of action. The Perform instructions allow testers to perform a dump on a tool and an optional part of the instruction is used to decode this dump in another tool. The syntax of the perform instruction is : `Perform a [Tool] dump (and decode it with [Tool])`. A problem arises with the semantic of the verb compute because this verb means that a change is performed on the data used for a test. We propose to testers team to tag these instructions by Step.

We take into account constraints to verify the coherence between started tools and stopped tools in a test block. First, we propose to the test user an auto-completion menu with the list of tools declared in the test chapter. Then, we verify that a tool that have been started is closed before the end of the test and that we cannot stop a tool not started.

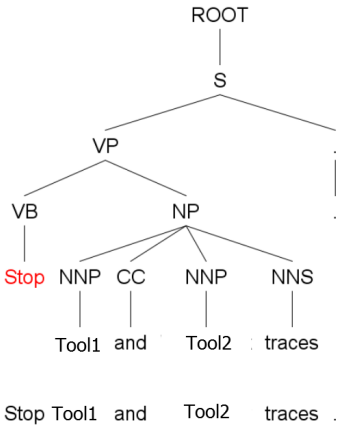


Figure 6: Syntax tree for Stop instructions

4.4 Languages composition and extension

The structure we provide for this specific language may be reusable for other specific domains of avionic systems. If we put away instructions that will be specific to a domain, the structure will be common for all domains. In order to manage different testing domains (network, cockpit command, flight command ...) specific instructions of each domain will be isolated from the structure. We next explain how we composed the language dedicated to the structure and languages containing instructions.

We use the language extension feature of MPS to make the link between structure language and instructions languages. As already seen in the Figure 4, the class hierarchy under instructions depends on which testing domain will be addressed. This class hierarchy is not part of the language which supports the structure. We are not even sure that instruction types (Reminder, Step, Check, Log and Trace) will be reusable unaltered in other domains. The MPS language composition feature offers an easy way to change the set of instructions without the need of implementing a new language from scratch. Then, we can now compose structure language with one or more instructions languages. Instructions languages must have concepts that extend the abstract instruction concept owned by the structure language.

5. RELATED WORKS

There are many test languages for software languages, but not so much for critical-embedded systems. We compare our language with some existing DSTLs. Then we explain on what our approach contributes to make processes more agile. And we recall the scope of our DSTL with Gherkin language used by Behavioral Driven Development methods.

5.1 Test languages

TTCN-3 [3] (Testing and Test Control Notation-3) is aiming on telecommunication protocols and distributed systems. It is an international standard, that also fits with the problem of real-time (hard and soft) and continuous systems.

This language is dedicated to test programmers and not for test engineers.

More generally a meta-model for tests of avionic systems in In-the-Loop context was proposed in the Guduvan thesis [7]. Its goal is to produce models in graphical or textual editors in order to be transformed into executable code with model-to-code transformations. The Model Driven Approach proposed in this context produces a complex language that could be difficult to use efficiently. Here, we do not want to create a language that covers all the needs of avionic systems, but we are focusing on keeping a small range of expressiveness in order to have a language easy to use. Targeted users of our language are only testers.

TDL [4, 20] (Test Description Language) is a language dedicated to the specification of functional tests. It will be used as a basis for the future implementation of executable tests on test frameworks. It supports a wide range of domain testing such as conformance tests, interoperability tests, distributed and concurrent real-time systems tests . . . Since test definition is platform independent, programmers only focus on the right test without taking care of how the test will run on the test bench. TDL is designed as a pivot language between user requirements and executable tests. Our language is concerned with the same problem. However, our language is dedicated to non-programmers and is scenario oriented while TDL is focused on executing tests through an abstract test bench. Tests in system integration phase are already complex, although we have restrict the scope of each language in order to increase usability and decrease learning time. We have chosen to provide a family of test languages close to the business domain that testers have to manage.

5.2 Agile processes for embedded systems

Other experimentations and researches come around introducing agile methods in embedded system context. If we consider the W-model and its testing activities, works we have found only focus on Software-in-the-Loop testing and hardware-software co-design testing. Best practices coming from eXtreme Programming (XP) and Scrum can be reused in embedded system development, but there are some difficulties to import some of them.

The first contribution addresses small hardware-software co-design projects [16]. They extend the CppUnit framework by providing digital signal processing possibilities and new assertions for speed and memory constraints. They demonstrate that unit test tools can be adapted to work in embedded environments. Other related works are only concerned by embedded software.

Works done by Salo and al. [15] aim to show the usefulness of XP and Scrum for embedded software developments by studying many agile projects from European organizations. The result of this study is that these organizations seem to be able to apply agile methods in their projects and report fairly positive results of their application. In this context, a process named Safe Scrum was proposed by Stålhane and al. [19] to introduce agile principles in a software certification context. Karlesky and al. [10] developed a pattern to test embedded software in using mocks for hardware components. They focus on testing software as soon as possible to help programmers to produce a code easily testable. Even if this proposition is attractive, a simulation of entire hardware is required for Software-in-the-Loop testing in our context.

A last proposition concerns the introduction of high-speed software engineering for embedded software [11]. They proposed to mix agile practices with classical process improvement activities. In particular, they introduce a test first approach in focusing on unit testing for embedded systems.

This is exactly what we want to try in Hardware-in-the-Loop testing activities. First, the need is to produce well suited unit tests in order to get more relevant tests. We hope that the tests do not uniquely serve for V&V purposes but also help to debug systems and bring the gap between integration tests and requirements.

5.3 Ubiquitous vs Domain Specific Language

A Domain Specific Language is very closed to an ubiquitous language using by the Behavior Driven Development (BDD) approach [18]. An ubiquitous language is a language whose structure comes from the business model. It contains the terms which will be used to define the behavior of a system. The main idea is that customers and developers share the same language without ambiguity. In the BDD approach, end users describe scenarios using the Given-When-Then pattern that generates test code skeletons that programmers must complete. Some frameworks assure the traceability from scenarios describing use cases to the implementation.

Even if this idea is attractive, multiple end users can provide a too wide language for a domain. DSLs are more restrictive and less flexible to minimize the complexity of the language. The glue code required by the BDD approach is encapsulated by the semantics of the DSL.

However, there are works using natural language processing tools that derive test case scenarios from a specification written in natural language [17]. We want to use the same approach but with a different goal. In our work, starting point is a precise description of test procedures in natural language. We want to produce a DSTL that covers all the need to write test procedures in a specific domain.

6. CONCLUSION

Our objective is to provide a first step in the production of a unified language for system testing integration procedures. This unification will be bring by a computer language that will fit with testers concern. Sharing a common language on a specific domain testing will facilitate the communication between tester teams and test programmer teams involved in tests production. Sharing a common language on a specific domain testing will facilitate the communication between tester teams and designer teams involved in systems design. This common language matches with the first value of the agile manifesto: "Individuals and interactions over processes and tools". Moreover, writing tests in a computer language rather than in a natural language implements the principle of "Working software is the primary measure of progress".

Moreover, paradigm of unit testing improves the structuration of integration tests. Concerning the ACOVAS project, the next step of our work will focus on a first instructions language dedicated to networks integration. We would like that testers teams adopt the language by rewriting test procedures in a well-suited environment. We would like to collaborate with test programmers teams to give an operational semantics on integration tests. We want to use a generic state-machine based execution environment in order to run some parts of test procedures.

To render the language design process less empirical, we would like to use supervised machine learning to find domain ontologies. These ontologies are the basis for concrete syntax for each avionic business. Concepts correspond to objects of our language and semantic links between concepts correspond to an action language on those objects.

7. ACKNOWLEDGMENTS

We would like to thank all people that participate to the ACOVAS project and especially testers that gave us feedbacks and helped us producing a language that fit with their business. We also would like to express our gratitude to Thierry Millan that helped us formalizing meta-model and producing OCL rules [13].

8. REFERENCES

[1] F. Campagne. *The MPS Language Workbench*, volume 1. FABIEN CAMPAGNE, 2013-2014.

[2] F. Campagne. *The MPS Language Workbench*, volume 2. FABIEN CAMPAGNE, 2015.

[3] ETSI. TTCN-3. http://www.ttcn-3.org/.

[4] ETSI. The Test Description Language (TDL); Part 1 : Abstract Syntax and Associated Semantics, 2015. ES 203 119-1.

[5] M. Fowler. Language workbenches: The killer-app for domain specific languages. http://www.martinfowler.com/articles/languageWorkbench.html.

[6] M. Fowler. *Domain-Specific Languages*. Addison-Wesley Signature Series (Fowler). Pearson Education, 2010.

[7] A.-R. Guduvan, H. Waeselynck, V. Wiels, G. Durrieu, Y. Fusero, and M. Schieber. A Meta-Model for Tests of Avionics Embedded Systems. *Proceedings of the 1st International Conference on Model-Driven Engineering and Software Development*, pages 5–13, 2013.

[8] Itemis. XText. *http://www.eclipse.org/Xtext/*.

[9] Jetbrains. Meta-programming system. *https://www.jetbrains.com/mps/*.

[10] M. Karlesky, W. Bereza, and C. Erickson. Effective test driven development for embedded software. In *Proceedings of IEEE International Conference on Electro information Technology*, pages 382–387, May 2006.

[11] P. Manhart and K. Schneider. Breaking the Ice for Agile Development of Embedded Software: An Industry Experience Report. In A. Finkelstein, J. Estublier, and D. S. Rosenblum, editors, *Proceedings of the 26th International Conference on Software Engineering*, pages 378–386. IEEE Computer Society, 2004.

[12] C. Manning, M. Surdeanu, J. Bauer, J. Finkel, S. Bethard, and D. McClosky. The stanford corenlp natural language processing toolkit. In *Proceedings of 52nd Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 55–60, Baltimore, Maryland, 2014. Association for Computational Linguistics.

[13] T. Millan, L. Sabatier, T.-T. Le Thi, P. Bazex, and C. Percebois. An ocl extension for checking and transforming uml models. In *Proceedings of the 8th WSEAS International Conference on Software Engineering, Parallel and Distributed Systems*, SEPADS'09, pages 144–149, 2009.

[14] A. Ott. *System Testing in the Avionics Domain*. PhD thesis, University of Bremen, 2007.

[15] O. Salo and P. Abrahamsson. Agile methods in european embedded software development organisations: a survey on the actual use and usefulness of extreme programming and scrum. *Software, IET*, 2(1):58–64, February 2008.

[16] M. R. Smith, A. K. C. Kwan, A. Martin, and J. Miller. E-TDD - embedded test driven development a tool for hardware-software co-design projects. In *6th International Conference, XP 2005, Sheffield, UK, June 18-23, 2005, Proceedings*, pages 145–153, 2005.

[17] M. Soeken, R. Wille, and R. B. Drechsler. Assisted behavior driven development using natural language processing. *Lecture Notes in Computer Science*, 7304 LNCS:269–287, 2012.

[18] C. Solís and X. Wang. A study of the characteristics of behaviour driven development. *Proceedings - 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 383–387, 2011.

[19] T. Stålhane, T. Myklebust, and G. Hanssen. The application of safe scrum to IEC 61508 certifiable software. In *11th International Probabilistic Safety Assessment and Management Conference and the Annual European Safety and Reliability Conference 2012, PSAM11 ESREL 2012*, volume 8, pages 6052–6061. Curran Associates, Inc., 2012.

[20] A. Ulrich, S. Jell, A. Votintseva, and A. Kull. The ETSI test description language TDL and its application. In *MODELSWARD 2014 - Proceedings of the 2nd International Conference on Model-Driven Engineering and Software Development, Lisbon, Portugal, 7 - 9 January, 2014*, pages 601–608, 2014.

[21] E. Visser. Spoofax. *http://strategoxt.org/Spoofax*.

[22] M. Völter, S. Benz, C. Dietrich, B. Engelmann, M. Helander, L. Kats, E. Visser, and G. Wachsmuth. *DSL Engineering Designing, Implementing and Using Domain-Specific Languages*. dslbook.org, 2013.