Abstract

The world of in-the-loop testing of avionic embedded systems is heterogeneous, consisting mostly of in-house test languages. A synthetic view of the different features possessed by these languages was deemed necessary, in order to allow reflections towards some homogenization. We considered a sample of six languages. Four of them are currently employed in the field of avionics. The other two, used respectively in the automotive and telecommunications industries, have been chosen for comparison purposes. We begin with the presentation of the specificities of the in-the-loop testing activity and the related needs for test languages. Afterwards, we discuss the different test languages in our sample, focusing on the features that they possess and how they fulfill the needs. We end the article with a proposal for a model-driven solution, homogenizing the different industrial test practices into domain-specific models from which test code is generated.

Keywords: test language; test execution automation; in-the-loop testing; test model; avionic embedded system

1. Introduction

An avionic embedded system is typically a distributed system, including several hardware components, such as: interconnected processors, memory modules, input / output cards, power supply devices, and others. The functional logic is mainly implemented by the software components running on the processors. The verification and validation of a system involves a rigorous testing process [1], where a series of test phases accompany its development, production and maintenance.

The hardware testing of avionic equipment represents the historic field of activity of Cassidian Test & Services (ATEC™ Series 6 – [2], [3]). With the increase of the usage of software over hardware for the functional logic, as well as the growth in complexity, the field of activity has been enlarged to encompass full systems, tested in-the-loop with a model of their environment. Cassidian Test & Services already has a number of products and services relevant to this field (U-TEST™ Real-Time System - [4]), and wishes to consolidate and extend its offer. A current concern is the offer in test languages.
In hardware testing of avionic equipment, standardized test languages exist, such as ATLAS (Abbreviated Test Language for All Systems, [5]) or ATML (Automatic Test Mark-up Language, [6]). In contrast, the in-the-loop testing activity of avionic embedded systems involves predominantly in-house solutions. Proprietary languages have flourished, and there is now a need to gain a synthetic view of the language features that emerged in practice.

This paper aims at giving such a view, by analyzing a sample of existing languages. It is intended as a first contribution to on-going reflections in the avionic community, toward some homogenization – if not standardization yet – of industrial practice. The analysis was also useful for Cassidian Test & Services to determine its own research directions.

We begin Section 2 with a discussion on the specificities of the in-the-loop testing activity. From these we derive a set of capabilities that test languages have to provide.

Section 3 presents the set of languages that we analyzed, which includes four proprietary languages from the avionic domain. One of these languages is currently deployed on test platforms provided by Cassidian Test & Services ([4]). Because of confidentiality issues, we could not disclose the names of the other three. For the sake of comparison with other domains, we also considered two additional languages, respectively coming from the automotive and telecommunications industries: TestML ([7]) and TTCN-3 (Testing and Testing Control Notation Version 3, [8]).

In Section 4 we present the different features of the test languages in our sample, while in Section 5 we discuss how these features impact the desired capabilities.

Section 6 states the directions for future work based on our results. We propose that the multiplicity of proprietary solutions be addressed by a model-driven approach with code generation facilities.

2. Specificities of In-the-Loop Testing

Figure 1 provides a schematic view of the life-cycle of an avionic embedded system. After the system specification and design phases, software and hardware components are developed and individually tested, before software and hardware integration testing proceeds. At the end of the development process, the target system – together with additional avionic embedded systems and with hydraulic, mechanical and other systems – is embedded into an aircraft prototype (Iron Bird). Later, a flight test program is performed. Once certification has been passed, production and maintenance processes are entered. The application logic does not need further functional validation, but hardware testing activities are still necessary to reveal manufacturing and aging problems.

This paper focuses on in-the-loop testing phases that occur during the development process. We introduce below the specificities of in-the-loop testing, as they have an impact on the languages analyzed in this paper.

2.1. The Various Forms of In-the-Loop Testing

An avionic system is tightly coupled to its environment. Testing the functional logic requires producing a large volume of data that, in the operational environment, would come from other avionic systems and physical sensors.

In-the-loop testing addresses this problem by having a model of the environment to produce the data. The model of the environment receives the outputs of the system under test (e.g. commands to actuators) and computes the next inputs accordingly. In the case of a control system, computation must typically account for the physical rules governing the dynamics of the controlled aircraft elements.

As shown in Figure 1, in-the-loop testing comes in various forms: model-in-the-loop (MiL), software-in-the-loop (SiL) and hardware-in-the-loop (HiL).

MiL testing is performed at the early phases of system development: neither the software, nor the hardware components exist yet, and the tested artifact is a model of the system.
In SiL testing, the actual software is considered. Re-targeting occurs when the software is compiled for a different hardware than the target one (e.g. using a desktop compiler). Re-hosting is preferred for better representativeness: the binary code is the same as the one in the actual system, and it runs on an emulator of the target hardware.

Finally, HiL testing uses the actual software running on the target hardware.

![Fig. 1](image-url)  
**Fig. 1** The life-cycle of an avionic embedded system and the existing types of testing activities

For complex systems, the MiL / SiL / HiL classification might be too schematic. Hybrid forms of in-the-loop testing can be considered, where the tested artifact includes system components having different levels of development. For example, one component is included as a model (MiL), while another one is finalized (HiL). Integrating components with such different levels may however raise difficult interconnection and timing issues.

### 2.2. Stakeholders

Historically, the aircraft manufacturer was in charge of all the integration activity for avionic embedded systems. It received components from the aircraft equipment provider. Then it integrated these into systems, until complete multi-system integration within the aircraft.

Nowadays, there is a shift of activity from the aircraft manufacturer towards the equipment providers, as the latter are asked to participate in the first integration phase. Thus, the aircraft manufacturer would now directly receive an integrated avionic embedded system: the equipment providers are becoming system providers. When looking at Figure 1, the horizontal line delimiting the intervention of the providers has a tendency to move upward.

The aircraft manufacturer historically has the needed expertise for setting up the in-the-loop testing activity. This activity, now becoming the responsibility of an avionic system provider, opens an opportunity for collaboration between the two. A new type of interaction emerges, comprising the exchange of information on the tests that were
executed by the system provider and the aircraft manufacturer. The exchange could concern test specifications, test procedures implementing the test specifications, or test data traces monitored during the execution of the procedures.

Naturally, each actor has its own internal tools and test platforms that it uses for testing. Inherent incompatibilities between them severely limit the exchanges that can be done. In practice, a test cannot be easily ported from one environment to the other.

The difficulties encountered during these new types of interactions between the different stakeholders have motivated our analysis.

2.3. The Interfaces of the System under Test

In the field of avionics, the interfaces of an avionic embedded system are more or less formally presented in an Interface Control Document (ICD). This name is generic and does not define a standard. Each enterprise is free to define its own ICD format, or even different formats for different aircraft programs. Whatever the specific format, the document contains information on the interfaces at several hierarchical levels (Figure 2), similar to those present in the OSI model.

At the lowest level, that of physical connections, the connectors of the system under test are presented and given unique identifiers. The pins of each connector are presented as well. Afterwards, the buses and lines that pass through the physical connections are indicated. At a low logical level, the messages are mentioned. Finally, at the highest logical level, the application parameters and signals are described. These represent the data used and produced by the embedded software. A signal corresponds to an instance of an application parameter encoded on a specific bus. For example, the aircraft speed can be sent to two neighbors, using respectively an AFDX connection for the first and an ARINC 429 for the second.

Several types of system network elements are used in the field of avionics for the communication between components, such as the following communication buses:

- Discrete,
- Analog,
- AFDX (Avionics Full-Duplex Switched Ethernet),
- ARINC 429 (Aeronautical Radio, Incorporated),
- MIL-STD-1553B,
- …

For example, let us assume that an avionic embedded component possesses on its interface a connector with a pin conforming to the ARINC 429 standard. This pin is used, naturally, for an ARINC 429 bus. In turn, the ARINC 429 bus communicates several ARINC 429 labels, where each label determines the set of application parameters that constitute the payload of the message. One of these parameters could be the speed of the aircraft. Figure 2 shows what a corresponding ICD would look like.

As mentioned before, the information is organized in a hierarchical manner inside the ICD. There is a tree structure with connectors at the top and application parameters at the bottom. Because such parameters are functionally meaningful to avionics engineers, they are often called engineer variables. We will refer to them as such in the rest of this paper.

The ICD can contain additional information to that presented in the example, like the data type of the engineer variable, its maximum and minimum values, the encoding that was used, or its value refresh rate. As many in-house formats of ICD exist, the supplied information at the various levels can be more or less detailed. In this paper, we assume that the available information is sufficient for a target perimeter of tests.

In a system, several instances of a same engineer variable can be present. For example, such is the case when a component produces an engineer variable that is consumed by several neighboring components. Note that the corresponding interfaces can be of different types. Also, the component producing the parameter may be duplicated within the system for redundancy purposes.
Fig. 2  Interface Control Document (ICD) Example

Usually a *tuple* is used in order to *uniquely identify* a particular instance of an engineer variable. In the above example of speed variable, a tuple could represent a path in the tree-like structure of the ICD, containing for example the SYSTEM_NAME, the BUS_NAME, the LABEL_NAME and finally the SIGNAL_NAME:

"SUT_1/ARINC_429_1/LABEL_1/AC_SPEED_429".

Some information in the path is redundant because of methodological constraints (e.g. the name of each signal on a bus is unique). Hence, the long identifier seen above can be shortened. A triplet usually suffices. By eliminating the LABEL_NAME which is no longer useful, we obtain the following short identifier:

"SUT_1/ARINC_429_1/AC_SPEED_429".

Even short identifiers yield long names for engineer variables. In test procedures, it is often convenient to have aliasing mechanisms with symbolic names.

### 2.4. The Test Platforms

A simplified *test platform* for an avionic embedded system typically has the following components (Figure 3):

- the test controller,
- the test resources,
- the test network,
- and the test language.

The test controller is responsible for the automated execution of a test, written in some supported test language. As execution proceeds, commands are sent to test resources that perform the actual interactions with the system under test. The test network has two portions, one linking the test controller to the test resources (the *test control network*) and another linking the test resources to the system under test (the *test interaction network*). By means of the test resources and test interaction network, some communication points of the system under test are made accessible. Other resources may be needed for purposes different from communication, such as ventilation units for controlling the temperature of the system under test.
The ICD defines all communication points that could be made accessible for HiL testing. A given HiL platform thus provides access to a subset of ICD elements, with some control or observation capabilities attached to them. It may correspond to a black-box or a grey-box view of the system (e.g. a grey-box view where a bus connecting two internal components is accessed). Test resources include hardware devices such as network cards.

MiL and SiL testing can involve access points that are not in the ICD. For example, MiL testing typically uses an additional interface to control the execution of the system model (e.g. start, pause, resume, and so forth) or even its internal state (e.g. force an error state).

Conversely, some actions on ICD elements may have no counterpart in MiL / SiL test platforms. For example, if the tested model only knows about abstract engineer variables, bus message corruption actions are meaningless. In this case, test resources are software resources; there is no need for a test network if everything runs on one desktop computer.

As can be seen, there is a strong heterogeneity of the world of test platforms, depending on the MiL / SiL / HiL testing activity in which they are used. Some test actions are inherently not portable because they are too high-level or too low-level for the tested artifact. Other test actions should be portable, like reading or writing engineer variables. Whether they are easily portable depends on the ability of the test language to offer platform-independent abstractions, so as to hide the usage of test resources.

2.5. The Cyclic Behavior of Avionic Embedded Systems

Components of an avionic system typically exhibit a predominantly cyclic behavior, where an execution cycle reads the input data and computes the output ones. For example, the value of the speed of the aircraft is sent by the producing component to the consuming ones periodically, with a period in the range of several milliseconds.

The consuming components expect to receive this data within the time constraints imposed by the cyclic communication. They enter a specific error state if the communication with their environment does not respect the time constraints, usually within some tolerance. For example, the component would not enter the error state on the first
violation of the time constraint (i.e. a parameter arrives outside of its expected reception time window, or does not arrive at all), but only if this violation is repeated for a number of successive cycles.

Despite some tolerance, the system functionalities cannot be exercised unless all input data from the environment are produced when expected from the components. As already explained, this is the motivation for in-the-loop testing: the system under test is coupled to a model of its environment. In Figure 3, the test controller would manage the model of the environment.

It must be understood that the system under test and the model of its environment together form a (cyclic) closed-loop system. The data that the system needs is already produced internally, by the components of the system under test or by the model of the environment. As such, a specificity of in-the-loop testing is that test control actions mostly consist of modifications of already existing data. For example, data in transit over some system network link is intercepted and afterwards removed or written in a modified form. The data manipulation may expand several cycles. Asynchronous data can also be produced by some components or inserted during the test execution phase for fault-injection purposes. For this type of interactions with the system under test, information on the sequencing of the transmission of the different data is important (i.e. the insertion of an asynchronous message between two regular cyclic messages). Overall, in-the-loop testing yields a completely different approach than the one used for testing open-loop asynchronous distributed systems, where the asynchronous sending of a few messages triggers the functional activity of an otherwise quiescent system.

2.6. Usability of Test Languages for in-the-loop testing

The specificities of on-the-loop testing have an impact on the test languages that support this activity. The languages must provide an adequate vocabulary for the interfaces described in the ICD and for the interactions via these interfaces, including timed data manipulations. The provided vocabulary may depend on the targeted form of in-the-loop testing. Some languages will be specialized for a form of testing, for example MiL testing, while others will have the vocabulary to address a larger perimeter of tests.

For best readability and writability, it is desirable that the vocabulary be sufficiently high level to allow a concise description of tests. Readability is all the more needed as different stakeholders may need to exchange tests. It is important that engineers or operators can easily understand a test written in a language. Writability refers to the ease of defining a test in the test language. It affects both the productivity of test engineers and the quality of the test code, because poor writability is error prone. Writability is all the more important as test engineers may be expert in avionics systems and not in programming languages.

Portability across different platforms is also desirable. As already mentioned, low portability severely limits the exchanges that can be done between different stakeholders. Independently of test exchange problems, portability needs may arise from the obsolescence management of test platforms. For example, a new type of desktop computer is used as a test controller (with a different type of processor and operating system) and the test resources are provided by a new company (with different functionalities and drivers). Cross-platform portability depends on the ability of the language to abstract away from the implementation level.

After this introduction to the desired capabilities, we come to the examples of languages and their analysis.
3. Sample of Test Languages

Table 1 gives an overview of the chosen sample of test languages. The sample consists of:

- four proprietary languages from the avionic domain, which shall be named PL₁, PL₂, PL₃ and PL₄,
- TestML from the automotive domain,
- and TTCN-3 (Testing and Test Control Notation Version 3) from the networking and telecommunication domain.

<table>
<thead>
<tr>
<th>Test language</th>
<th>Industrial domain of use</th>
<th>Types of testing activities</th>
<th>Types of testing sub-activities</th>
<th>Based on existing language</th>
<th>Specification</th>
<th>Compiled / Interpreted</th>
<th>Standardization status</th>
</tr>
</thead>
<tbody>
<tr>
<td>PL₁</td>
<td>Avionics industry</td>
<td>In-the-loop testing</td>
<td>Model / Software / Hardware-in-the-loop</td>
<td>(OOPL: C++)</td>
<td>Use of libraries</td>
<td>Compiled</td>
<td></td>
</tr>
<tr>
<td>PL₂</td>
<td></td>
<td></td>
<td>Model-in-the-loop</td>
<td>✓ (OOPL)</td>
<td>Modification of the grammar / Use of libraries</td>
<td>Interpreted</td>
<td></td>
</tr>
<tr>
<td>PL₃</td>
<td></td>
<td></td>
<td>Hardware-in-the-loop</td>
<td>✓ (HSPL)</td>
<td>Use of libraries</td>
<td></td>
<td></td>
</tr>
<tr>
<td>PL₄</td>
<td></td>
<td></td>
<td>Software / Hardware-in-the-loop</td>
<td>-</td>
<td>PL₄ grammar</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TestML</td>
<td>Automotive industry</td>
<td></td>
<td>Model / Software / Hardware-in-the-loop</td>
<td>-</td>
<td>XML Schemas</td>
<td></td>
<td></td>
</tr>
<tr>
<td>TTCN-3</td>
<td>Networking and telecomm.</td>
<td>Distributed systems and communication protocols testing</td>
<td>-</td>
<td>-</td>
<td>TTCN-3 grammar</td>
<td>Compiled</td>
<td>✓ [9]</td>
</tr>
</tbody>
</table>

Table 1: The chosen sample of test languages

The four proprietary test languages, from PL₁ to PL₄, have been chosen because they represent languages currently employed in the avionics industry. The first one represents the offer of Cassidian Test & Services on the U-TEST™ Real-Time System integration test platform ([4]). To the best of our knowledge, no public test language exists that shows all the characteristics exhibited by these four, and as such, their inclusion was deemed necessary. The fact that we cannot disclose some information does not have a strong impact on this paper, as our interest is to discuss general concepts and features of test languages. In the discussion, we will feel free to use examples of pseudo-code. They will not disclose the precise syntax of proprietary languages but suffice to capture the essence of the identified features.

For comparison purposes, the sample also includes two languages outside the field of avionics.

TestML is issued from a research project in the automotive domain. Its aim was to investigate the design of an exchange language, in the sense shown by Figure 4. The multiplicity of proprietary languages yields the need for many language translators, but if a common exchange language is used then the number of required translators is reduced. TestML is the only language of our sample that is not operationally used in the industry. It is a research product and its connection to proprietary languages is not implemented. However, it represents an attempt to synthesize concerns arising from the in-the-loop testing practice, so that its consideration was deemed relevant to us.
As shown in Figure 1, the test languages from PL$_1$ to PL$_4$, and TestML, exemplify the various forms of in-the-loop testing of embedded systems. TTCN-3 serves a different purpose, being used in the testing of distributed systems and communication protocols. It is included in our sample because avionic embedded systems are also distributed systems, and their communication is based on interfaces conforming to a wide range of communication protocols. It is thus interesting to look at best testing practice in the field. The maturity of TTCN-3, which is an international standard and at its third release, justified its consideration.

From the six test languages in Table 1, three are built upon existing general-purpose programming language, while three have been defined from scratch. PL$_1$, PL$_2$ and PL$_3$ fall in the first category. They are based on object-oriented (OOPL) or high-level scripting (HSPL) programming languages. For example, C++ and Java are of the OOPL type, while Python is a HSPL. Given a general-purpose language, two options exist to build the test language:

- the modification of the grammar,
- or the definition of specialized libraries.

Both options appear in Table 1. Test procedures in PL$_2$ require some pre-processing before they are fed to the interpreter of the language on which they are based, because the grammar is modified. Test procedures in PL$_1$ and PL$_3$ are processed by a standard version of compiler / interpreter.

PL$_4$, TestML and TTCN-3 have been specified from scratch. PL$_4$ and TTCN-3 are imperative languages with their specific grammar and compilation / interpretation tools. A standard implementation is defined for TTCN-3 in [10]. This implementation is typically done either in C++ or Java. TestML is a markup language based on XML Schemas, and uses automata descriptions to capture the behavioral part of the test. It is not an executable language; although a MATLAB Simulink (with the extension Stateflow) implementation of the automata has been proposed to demonstrate the semantics.

4. Analysis of the Sample of Test Languages

Our analysis considers four broad categories of language features:

- the organization of the tests (§4.1)
- the abstraction for test interfaces and the link to the architecture of the system under test (§4.2)
- the language instructions for performing test control and test interaction actions (§4.3)
- and the management of time (§4.4).

Their presentation in this section intends to remain factual. Afterwards, in Section 5, we will discuss how the identified features impact the level of usability of a test language.
4.1. Organization of the Tests

Figure 5 gives an overview of the organization types that can exist for test description artifacts.

**Intra-test organization** refers to the structured description of a test. **Inter-test organization** refers to the organization of individual tests into higher-level containers *(i.e.* sets of tests).*

We focus here on intra-test organization, because inter-test organization is typically managed by external tools. For example, a *test manager* (or *test director*) is in charge of the organization of tests into structured *test groups* where all tests in a test group share a specific quality *(e.g.* they concern the same system under test, the same functionality, the same detailed requirement). A *test sequencer* controls the organization of tests into *test suites,* where the execution of tests is ordered in a specific manner *(e.g.* using a test execution tree). Inter-test organization is thus not a major concern for the test languages, while intra-test organization is.

![Test organization types](image)

---

A simple form of intra-test organization is the capacity to factorize reusable code fragments into *functions* (or procedures, or class methods). All languages possess this feature, except for TestML that does not resemble a programming language.

Another form is to allow the regrouping of similar types of instructions within *test sections,* in relation to the different steps that are performed during the testing activity. Being a markup language, TestML has this form of organization but is the only language of the sample that does. TestML has markup elements in order to *stimulate* the system under test, *capture* data and *evaluate* its behavior.

Other useful sections could concern *test identification* information *(e.g.* the name of the test, the target system under test, the author of the test), *pre-test actions* *(e.g.* powering up the test platform, informing the test operator, powering up the system under test, reaching a desired state) or *post-test actions.* None of the analyzed languages make them appear explicitly as distinct sections of a test. It means that their identification has to rely on the methodology for writing tests *(e.g.* code annotation with comments delimiting sections, grouping of pre / post-test actions into separate functions).

The last form of intra-test organization we consider is very important for addressing complex test architectures. It consists of the possibility of having several active *parallel test components,* each possessing an independent execution flow. All languages of the sample offer this possibility, but there are discrepancies on what is concretely offered. PL3 has no specific notion of test component, but it offers the native multi-threading / multi-processing facilities of the language on which it is based. PL2 has a big main component and a set of small concurrent monitors with a behavior of the form: *condition* \(\rightarrow\) *action.* The action can be a log or a simple input stimulation. PL2 has commands to load and launch concurrent test scripts from another test script. PL1, TestML and TTCN-3 offer the richest notion of test component with a symbolic connection interface. TestML has concurrency at a high level of abstraction, in the embedded automata models. PL1 and TTCN-3 have concrete components concurrently executing their sequential code.
They both make it possible to have several instances of a generic component, each involving its connections with other component instances or the system under test. Figures 6 to 8 illustrate the concept.

In TTCN-3 (Figure 6), a test case always has a main test component (MTC). The MTC instance is implicitly created when the test case starts. Instances of auxiliary parallel components can then be explicitly created by the test case. The topology of connection is determined by means of connect and map operations, depending on whether the connection is with other components or the system under test. The start operation launches the execution of some behavior by the target component. Note how the behavior is passed as a function reference parameter.

```
TTCN-3

// The declaration of a test component possessing two ports

type component myMTCType_1 {
    myPortType_1 myPort_1;
    myPortType_2 myPort_2;
}

// A test case with an auxiliary test component
testcase myTestCase_1() runs on myMTCType_1 // type of the MTC as seen above
system mySUTType // access points to the SUT {
    // Creation of an auxiliary parallel test component
    var component myComp_1 := myCompType_1.create();

    // Connection between the MTC and a parallel test component
    connect(myComp_1.myPort_1, self.myPort_2);

    // Connection to the system under test
    map(self.myPort_1, system.port_1);
    map(myComp_1.myPort_2, system.port_2);

    // Start of the auxiliary parallel component
    myComp_1.start(myBehavior());
}
```

**Fig. 6** Test components in TTCN-3

In PL₁, generic test components are called user codes (Figure 7). There are also simpler (non-generic) monitors similar to the ones in PL₂, except that their action can also be the start of another component. PL₁ has no distinguished main component: from the point of view of the run-time, all user code instances and monitors are “equal citizens”, whose concurrent behavior yields a test case. A subset of them will be active at the beginning of the test; the other ones will be started by the actions of the active subset. Libraries are provided to control the execution of components (see userCodeToolkit in Figure 7). The language environment offers a graphical user-interface (GUI) to develop monitors and user codes; with code generation facilities (code skeletons, in the case of user codes). The GUI also allows for the declaration of user code instances. The overall topology of connection is saved in an XML configuration file (Figure 8). The topology description is thus not part of the source code of components. Neither the connect nor the map commands are present in PL₁, as the run-time interprets the XML file to build the configuration.

The next section further explains how the various languages manage the link with the system under test.
// A user code has an interface and a behavior
// The interface code is automatically generated from its
// specification via the GUI

class myInterface: public userCodeInterfaceBase {
    public: // Interface definition
        variableType_1 inputVariable_1;
        variableType_2 outputVariable_1;
    // Methods to initialize and handle the interface
    }

// A skeleton for the behavior is also generated

class myBehavior: public userCodeBehaviorBase {
    private:
        smartPointer<MyInterface> myInterfacePointer;
    public:
        executionResult step() {
            // *** The test behavior must be placed here ***
            myInterfacePointer->outputVariable_1 = 1;
            userCodeToolkit.startInstance("instanceName2");
        }
    }

Fig. 7 User codes in PL₁

Fig. 8 XML topology configuration for the PL₁ runtime

4.2. Link with the System Under Test

All languages of the sample have a logical view of the system interface, abstracting away from the concrete access points. We saw the example of TTCN-3 in Figure 6. The system under test is viewed as a specific system component offering typed communication ports. Test component ports can be dynamically mapped to the system logical ports, yielding a dynamic test architecture. The TTCN-3 executable uses the services of a separate test adapter layer to implement the concrete communication operations. It makes the test procedures portable and independent of the test platform implementation. The multi-layered architecture of a TTCN-3 test system is standardized, with well-defined interfaces between layers.

The test languages for in-the-loop testing also hide the implementation details of the platform, although not in a standardized way. Their system logical interface consists of engineer variables (all languages) and possibly lower-level ICD elements (PL₁, PL₃ and PL₄). Languages focusing only on engineer variables are targeting MiL testing (PL₂), or intend to describe test procedures meaningful whatever the MiL / SiL / HiL form of testing (TestML). The other languages seek to also address the specificities of HiL testing, hence the visibility of ICD elements like buses or messages.

None of the languages exhibits the dynamic mapping facility of TTCN-3. The test architecture is always static, as is also the architecture of the system under test. When test components have a symbolic interface (PL₁, TestML), the mapping with the logical system interface is described once and for all in an XML-based format (see Figure 8).

All languages for in-the-loop testing have a notion of dictionary of available engineer variables. In the avionic domain, the names in the dictionary are usually built by parsing the tree-like structure of an ICD document. A name may include
the complete tuple (the *long identifier*) or a triplet (the *short identifier*) that suffices to uniquely identify the variable (for more details see Section 2.3). Aliasing mechanisms may be provided in order to further shorten names.

How a source code uses a name to access a variable varies from one language to the other. In PL₁, the dictionary is an external service to which data accessors are requested. At the opposite, PL₃ has a global dictionary structure directly accessible in the execution environment of the test. PL₂ and PL₄ represent intermediate language design choices. The source code does not explicitly handle accessors, but a specific symbol (PL₂) or an access declaration instruction (PL₄) indicates the special access. Figure 9 summarizes the various possibilities.

```plaintext
PL₁

// An accessor to the linked variables is given at the creation // of the user code interface object. Other variables can also // be accessed, but then the user code behavior needs to ask // for an accessor.

variableToolkit.initLink("SUT_1/ARINC429/LABEL_1/AC_SPEED_429");

myAccessor = variableToolkit.getAccessor("ICD_NAME_1/ARINC429/LABEL_1/AC_SPEED_429");

myAccessor.set(0); // Writes the variable
myAccessor.get(x, &timestamp); // Reads it in x

// More complex test actions use either the name:

variableToolkit.testAction("SUT_1/ARINC429/LABEL_1/AC_SPEED_429");

// or the accessor as a parameter:

otherToolkit.otherTestAction(myAccessor);

PL₂

// Access is granted to any engineer variable, it is denoted by // a special character ‘@’ for the test language pre-processor

x = @modelSUT_1/AC_SPEED;
@modelSUT_1/AC_SPEED = 0;

// More complex test actions use the name of the // variable as a parameter, not the accessor

eToolkit.testAction("modelSUT_1/AC_SPEED");

// Access is also given to the status variables of the system // models (MiL testing activity)

isStatusRunning = @modelSUT_1/Running;

PL₃

// A dictionary data structure is provided as a global variable // in the execution environment. An alias set structure may be // defined to allow indexing by short names.

aliasSet = {
    ['SUT_1/ARINC_429_1/AC_SPEED_429'] = {
        alias = "myAlias"},
}

x = dictionary.myAlias;
dictionary.myAlias = 0;
testAction (dictionary.myAlias);

PL₄

// Access is gained by the declaration of the needed variables. // An alias may be introduced by the declaration instruction.

access engineerVariable "SUT_1/ARINC429/LABEL_1/AC_SPEED_429" as myAlias;

x = my Alias;
myAlias = 0;

aToolkit.testAction (my Alias);
```

Fig. 9 Access to system variables in the proprietary test languages (from PL₁ to PL₄)

Figure 10 exemplifies access to other ICD elements. It focuses on PL₁ and PL₄ because PL₃ offers much less accessibility than the former two. In general, buses and messages are accessed for fault injection at a low level. PL₄ is more oriented toward testing high-level functionalities of systems-of-systems, and its users preferably have external tools for low-level tests (e.g. a bus analyzer / exerciser).
In PL₁ and PL₄, users have the possibility to control both the functional activity and the faults. Bus and message names are built from the ICD document, similarly to what we saw for engineer variables. It is interesting to note that the structure of the ICD is only partially reflected by the languages. In PL₁, gaining a bus accessor does not automatically provide access to its messages, although the bus accessor is used to get a message accessor. In PL₄, the bus and message levels are kept completely separated. In both languages, the engineer variable level is separated from the other ones.

Having an abstract interface in terms of ICD elements allows the test description to be independent from test platform implementation details. The runtime of the various test languages interprets the abstract test actions into concrete test actions involving test resources. At a concrete level, variables are managed differently according to their sampling or queuing mode, which is specified in the ICD document. In the sampling mode, the data is sent or received periodically even if no explicit test action writes or reads it. In the queuing mode, the data is asynchronous (i.e. an abstract write action triggers the sending of a specific message). The runtime knows which test resource manages a given low-level ICD element (e.g. which communication card manages the communication bus on which a variable is sent). In the case of PL₁, the configuration is described in an XML file. For each category of test resource, the PL₄ runtime implementation uses a generic interface that hides the vendor-specific interface. We do not comment on the management of test resources in the case of the other test languages, as we did not have sufficiently detailed information.

### 4.3. Types of Test-Related Instructions

Test-related instructions include: test execution control, test interaction and test report instructions. We provide below a brief discussion of the control and report instructions, before focusing on the interaction with the system under test instructions.

We saw an example of test execution control with the `start` instruction in Figures 6 and 7. All test languages have instructions to control the execution of a test. The control may involve timing aspects such as a periodic activation of the test code. These aspects are relevant to stimulate cyclic embedded systems. We delay their discussion until Section 4.4, where we address the management of time. Besides parallel test components, execution control may also concern the system under test, but this is only for the MiL form of testing: PL₂ has specific instructions to control and synchronize the execution of the system models.

As regards test reporting, the most powerful built-in facility is provided by TTCN-3. It allows for the production of local verdicts and the synthesis of a global verdict from the local ones. Verdicts are ordered from `fail` to `pass` with rules enforcing a conservative direction of changes: a `pass` verdict may change to `fail` or `inconclusive`, but a `fail` verdict never changes. A form of verdict management is also provided by PL₂. The other languages do not put emphasis on verdicts, because test evaluation is usually not performed on-line. SiL and HiL test platforms include detailed data recording facilities, and the recorded data is analyzed off-line by test engineers with the aid of visualization tools.
Let us now focus on test interaction instructions. There are interesting differences between TTCN-3 and the test languages targeting embedded systems. TTCN-3 abstracts all interactions into a small number of core instructions: send and receive for message-based communication, call and reply for synchronous communication with remote services.

Remote calls have no equivalent in the other languages of the sample, because the target embedded systems do not implement this form of communication. They only have message-based communication. Yet, most test interactions are not defined in terms of sending and receiving messages. Rather, they are defined in terms of reading, writing or modifying the value of engineer variables. The variables are an abstraction for the data carried by messages. The underlying message-based interactions with the system under test are implicit in the test description; they are delegated to the test language runtime.

The most basic form of variable-based interaction is to read or write a value. For this, simple assignment is offered as a syntactic facility. We saw examples in Figure 9 for PL-3, PL-3 and PL-4. In PL-1, for which variable accessors are explicit, only a restricted form of assignment is provided. It involves a local copy of the variable and an automatic synchronization with the global one at fixed execution points. For example, in Figure 7, outputVariable_1 is synchronized with the global variable before and after each execution of the step method. In the general case where finer synchronization is needed, the PL-1 code uses the get and set methods of the variable accessors, not the assignment. Whatever the language, a write forces a value until another value is written or the writing is explicitly disabled. In the sampling mode, it means that the test platform keeps sending the value until a new instruction is provided.

In addition to the simple read and write, all in-the-loop testing languages of the sample offer a rich set of predefined variable-based interactions. They typically include stimulation patterns over time like ramp, oscillation, triangle or square patterns. Figure 11 exemplifies the ramp pattern where the successive values of the ramp are calculated from a start value. Other patterns depend on the current value of the variable, like the one injecting an offset.

```
// Using a formula interpreter
Formula formula = new Formula("2*@t + 1");
Stimulation stimulus = new Stimulation();
stimulus.define("modelSUT_1/AC_SPEED", formula);
stimulus.start();
```

```
variableToolkit.injection("aVariableName", "offset", offsetValue, listOfParameters);
```

Fig.11 Examples of variable-based interactions

PL-3, PL-3 and PL-4 have fault injection instructions not only for variables, but also for other ICD elements like messages and buses. In PL-3, the instructions are kept basic (e.g. stop any message emission on the bus) because external injection tools are used in complex cases. PL-1 and PL-4 allow for richer fault injection from the language, like modifying the content of a message or sending spurious messages. PL-4 has made the choice of offering generic injection libraries, while PL-1 has specialized ones according to the type of bus. Let us consider the example of message corruption. PL-
takes advantage of the knowledge of the encoding format (e.g. it offers an instruction to change a specific status bit in an ARINC 429 message). PL₄ sees messages as raw bit vectors and provides a generic instruction to overwrite a vector.

To conclude, languages for in-the-loop testing put emphasis on data manipulation rather than on communication primitives. They offer many predefined instructions to cover the recurring manipulation patterns. The consideration for fault injection at different levels of the ICD further adds to the number of instructions. We noticed some heterogeneity in the way the instructions are incorporated into the language. In one case, there is an overloading of a usual language operator (i.e. the assignment). In the other cases, the instructions are grouped into specialized toolkits, attached to ICD elements accessors, or passed as string commands to more generic instructions. In contrast, TTCN-3 has a homogeneous view of its communication instructions: they are all offered as methods of port objects, where a component port is strongly typed according to the messages or service calls that it can transmit. It has been proposed in [11] to add a stream port type, which would allow TTCN-3 to account for continuous data flows.

4.4. Time Management

Time is not a major concern for a language like TTCN-3. It addresses functional issues of distributed systems and merely offers basic timer operations using the local clock of components. Note that real-time extensions have been proposed ([12], [13]), with a time-stamping of communication events and a timed control of events.

For in-the-loop testing, time is a prevalent notion. Embedded systems process time-stamped data and typically exhibit execution cycles of predefined duration. We already mentioned that the test languages offer a number of time-dependent stimulation patterns (Figure 11). Data observation can also be made time-dependent, as in the following PL₁ instruction:

```
eventToolkit.waitValue(myAccessor, expectedValue, tolerance, timeout);
```

or in a similar PL₄ instruction:

```
eventToolkit.waitCondition(myCondition, timeout, checkPeriod).
```

Note that PL₁ and PL₄ run on top of real-time executive supports.

Usually, time is expressed in physical units (e.g. in seconds). In PL₂ and PL₄, we found some instructions with logical units (i.e. a number of cycles). This is quite natural for the MiL testing usage of PL₂, because the execution of the test components can be precisely synchronized with the execution of the models. The resulting test system can be simulated in a stepwise manner, cycle by cycle. Such synchronization is of course not possible for the other forms of in-the-loop testing. Rather, test control facilities are offered to make the execution of the test compatible with cycle durations in the target system.

Figure 12 shows the facilities offered by PL₁, PL₃ and PL₄. Depending on the language, timed test execution control is applied at a different level of granularity. The finest-grained control is in PL₄. It is at the level of blocks of instructions, where a test can contain blocks to be executed on tick reception from the synchronization service. A block is to be executed in bounded time. In PL₁, test execution control is at the component level. A user code instance is either asynchronous or periodic. The periodic activation comes with a control of the execution time, which must not exceed the period. PL₃ does not have the notion of component, but a test procedure can be repeated periodically. We did not include TestML examples in Figure 12. Compared to the others, this language would have a very different flavor due to the use of hybrid timed automata. Note that the automaton formalism inherently has the possibility to represent a precise timed control of test execution.

As can be seen, the forms of timed control are heterogeneous among the languages. It is usually not necessary to have a precise timed control over all test components. However, parts of the test will need to be aware of the time scales in the system under test and of its cyclic behavior. This is where facilities like periodic activation and bounded time execution prove useful.
// User code instances can be declared as periodic in the XML configuration file. If the execution time exceeds the period at runtime, an error will be issued.

<userCodeInstanceDeclaration ... period="100.0"/>

// A test can be periodic

testIdentification = {
    testPeriod = 100.0;
}

// Execution control is for blocks of test instructions. It uses a tick service.

tick.register(50); // Frequency is 50 hertz
tick.wait();      // Code to be executed upon reception of a tick.
tick.complete();  // An error is issued if another tick occurs before completion.

... // Asynchronous code

tick.wait();      // Code to be executed upon reception of some subsequent tick.
tick.complete();  // A pseudo periodic behavior can be expressed in a loop. Extra ticks may occur between two iterations.
while (logicalCondition) {
    tick.wait();      // Pseudo-periodic code.
tick.complete();
}
tick.stop();      // From now on, no tick synchronization.

Fig. 12 Timed control of test execution

5. Discussion on the Usability of Test Languages

In the previous section we have analyzed a number of features exhibited by the test languages in our sample. We now focus on their impact on language usability. We also make suggestions for additional facilities that would be useful. The discussion is organized by categories of features.

5.1. Types of Test Languages

Some languages were defined on top of existing general-purpose programming languages (from PL₁ to PL₃), while others were specified from the ground up. Among the latter specific test languages, we distinguish those that resemble imperative programming languages (PL₄ and TTCN-3) and more abstract ones (TestML).

Specific test languages have the upper-edge concerning readability and writability. They offer a vocabulary that is specific to the industrial context of usage and they are more concise. In GPPL-based languages, the domain-specific vocabulary is incorporated using native lower-level constructs that obfuscate the test code. In order to bridge the gap between a generic language and a specific one, test languages based on an existing GPPL rely on automatic code generation or on language pre-processing techniques. Code generation has a positive impact on writability, but only in the initial phase. When a test has to be analyzed or modified later, the user has to delve into the generated code. As such, a negative impact on readability is noticed. Language pre-processing should yield better readability, but this depends on the amount of pre-processing that is done. In practice, in the PL₂ example, many facilities are offered in libraries rather than in new syntactic keywords. The source code still exhibits the idiosyncrasies of the native language.

The comparison between TestML and the other languages raises the issue of the choice of the language paradigm. A distinguishing feature of TestML is its model-based paradigm with the choice of hybrid timed automata. We believe that the use of the timed automata abstraction, although clear and rich in its semantics, would not be a preferred choice in the field of avionics, as test engineers are accustomed to using more down-to-earth techniques. In our sample, all languages used in the avionics field have an imperative programming paradigm. For better readability and writability of the test code, it is definitely preferable to have a language at a higher level of abstraction than say, a GPPL. However,
for better acceptance by engineers, the proposed solution should accommodate the existing custom and practice. TTCN-3 can be seen as a good compromise in its domain of application, combining convenient high-level constructs and an imperative style familiar to engineers.

Concerning portability, no standardized implementation exists for the proprietary languages of the sample. In this context, test languages based on an existing GPPL are inherently more portable than specific ones. Those using a library-based approach are more portable than those modifying the grammar of the GPPL, as existing compilers / interpreters can be easily reused. In any case, portability across different platforms also depends on the implementation choices made inside the language runtime. A clear separation must be offered between a generic execution kernel of the test language and all platform-specific adapters, similar to their separation inside the standardized implementation proposed for TTCN-3.

5.2. Intra-test Organization Features

A test can be a quite complex artifact. Convenient intra-test organization features are thus crucially needed for both readability and writability of the code.

The use of functions, allowing for code reuse by means of a concise manipulation, is only a first facility. The definition of test sections is usually not offered by the languages, but could aid the test engineer to find the information s/he is looking for inside a test, having a positive impact on readability. Concerning writability, the separation of a test into several test sections would force the test engineer to enter all the needed data and to not forget important information (e.g. the initialization of a test).

The usage of multiple threads of execution cannot be avoided because of the complexity of the system under test and of the tests themselves. Predefined component constructs have the advantage of hiding low-level thread control functionalities. We have extracted two important ideas from our analysis of the existing constructs. The first one deals with the need for formal interfaces, so that multiple instances of a component can exist in the test architecture. The second one is the possibility to accommodate both complex and simple component constructs. The complex construct is the most general one, but it is convenient to also have a predefined monitor construct for the simple cases. It lets engineers easily express the observation of logical conditions and the reactions to trigger when these conditions occur. Having monitors in separate components also gives a better visibility to the important logical conditions of the test, hence improving readability.

5.3. Link with the System under Test

The fact that all test languages in our sample offer logical interfaces to the system under test, based on the analysis of its ICD, renders them somewhat platform-independent and thus positively impacts portability. In order to cover a large perimeter of tests, a test language must offer access to the elements present at various hierarchical levels of the ICD.

We observed a great deal of heterogeneity concerning the way access is offered, sometimes for a same hierarchical level and sometimes between different levels. For example, PL_1 and PL_2 offer two ways to access engineer variables, depending on the desired test action: sometimes by means of accessors, sometimes by means of a name passed as a string parameter to the action. Needless to say, the first approach is cumbersome unless syntactic facilities are provided to hide the handling of accessors. The latter approach with strings has the drawback that static type-checking is impossible, which may be error-prone. The checks can only be performed at execution time, when the test action processes its parameters. PL_3 offers an interesting solution devoid of explicit accessors and string parameters: all engineer variables are directly accessible as global variables in the test execution environment. This facility is however only for engineer variables. When descending at lower levels of the ICD hierarchy, we once again find the string parameter solution.

Having a heterogeneous view of ICD elements is detrimental to both readability and writability. It would be preferable to have a consistent access policy that spans all ICD levels. The string parameter solution should be avoided to allow for a strongly-typed language. It would be preferable to have typed access via global objects for all ICD elements, variables
but also buses and messages. The structure of the ICD could even be reflected into the structure of the objects, allowing explicit navigation into the hierarchical levels.

5.4. Offered Interaction Instructions

Test engineers are offered a high number of interaction instructions, possibly covering different levels of the ICD hierarchy. All test languages have tried to regroup the instructions in a structured manner. Some languages regroup instructions into toolkits, with each toolkit being specific to a hierarchical level in the ICD or to a specific type of test actions. Others link the different instructions to the accessors of ICD elements. These two solutions are mostly used for simple types of instructions. More complex instructions are usually not found inside the language and are passed as a string command parameter to some generic toolkit. This augments error-proneness as no syntactic verification of the command is possible at compilation time.

Again, the heterogeneous view of interactions inside a language negatively impacts the readability and writability of the test code. There is a need for a coherent organization of instructions, which also allows for type checking at compile time. Such a coherent organization is provided by TTCN-3, where interaction methods are attached to typed port objects. The principle of attaching test actions to typed interface objects can also be retrieved in other industrial contexts, like Graphical User Interface (GUI) systems (classical applications in [14] and web applications in [15]): an application window possesses several buttons, each button has a number of test actions attached (e.g. click), etc. Similarly, we could have methods attached to ICD elements, where each type of elements would call for its specific test actions.

5.5. Management of Time

Time is a required concept for languages addressing in-the-loop testing of embedded systems. At a low-level, the runtime must be able to manage sampled data; this is usually transparent to the test code. Explicit account for time is offered by two means. First, the languages have interaction instructions with time parameters, like the generation of a ramp over several cycles or the observation of a system reaction to occur in bounded time. Second, the user has the possibility to explicitly control the execution of pieces of code using clock cycles. Depending on the language, we found a different granularity of control, from blocks of instructions to entire tests.

We believe that periodic execution should preferably be controlled at the granularity of components. Complex tests involve both periodic and asynchronous actions; hence we need a finer granularity than the test. On the other hand, code mixing asynchronous and periodic pieces is less readable than code separating them into coherent execution units.

Finer-grained control may be useful for aperiodic components with blocks of instructions to be sequentially executed at consecutive cycles. None of the language offers this form of control. In particular, the control instructions of PL4 do not offer it: they do not guaranty that there is no tick between the end of a synchronous block (tick.complete) and the beginning of the next (tick.wait). We think that the time-triggered sequential execution could be a useful facility to add to the languages. Moreover, the semantics of high-level instructions like ramps could be defined in terms of a time-triggered execution of lower-level instructions, yielding a consistent way to integrate the stimulation operators into the languages.
6. Conclusion and Perspectives

In this article we have analyzed six test languages. We focused on four proprietary languages (from PL₁ to PL₄) that are currently employed in the avionics industry, for the in-the-loop testing of avionic embedded systems, at different integration (i.e. component, system and multi-system) and maturity (i.e. model / software / hardware-in-the-loop) levels of the system under test. For comparison purposes, we also looked at a test language issued from a research project in the automotive industry (TestML) - covering the same type of testing activity, as well as a mature international standard (TTCN-3) used for the testing of distributed system in the field of networking and telecommunications.

Our analysis focused on a number of features and the way they are offered by each test language. We identified a high level of heterogeneity among our set of test languages: not all test languages offer the same features and shared features are offered in different manners. It would thus be difficult to choose one of the proprietary test languages from our list, slightly improve it and retain it as a standard for the avionics domain. Actually, we now strongly believe that the multiplicity of in-house solutions should be addressed at a higher-level, the one of language concepts and of test design models. We envision a model-driven approach where test models are developed, maintained and shared, and are then automatically translated into target (possibly in-house) executable languages. The model becomes the central entity of the test development activity, replacing current approaches where the code occupies this position. The shift is driven by the perception that test software is indeed software, and that test development can benefit from advanced software engineering methodologies [16], such as meta-modeling techniques and model-to-code transformations. There is currently active research on the introduction of modeling technologies into test development, among which we can mention the UML-based modeling of simulation software for MiL testing in [17], or more fundamental work to produce TTCN-3 code from models complying with the UML Testing Profile [18]. In a different industrial domain, that of web applications, [15] proposes a model-driven approach for the definition of platform-independent test models complying with their own UML Profile. Our future work intends to contribute to this field of research, transferring advanced methodologies to the development of test artifacts. It will build upon our analysis of the existing languages.

We wish to develop a meta-model of in-the-loop tests that captures the language concepts we identified as of interest. The meta-model should cover a wide perimeter of tests, from MiL to HiL. Concepts of interest include: test sections, parallel test components with formal interfaces, simple monitor components, syntactic facilities to access a hierarchy of ICD elements, a rich set of test actions attached to typed ICD elements, and time management with various forms of control of test execution. Their integration is a challenging issue, as all these different aspects must be taken into account in a coherent manner. One of the problems observed in the test languages was that the addition of new functionalities tended to lack homogeneity. We will need to formalize the different concepts in an easily upgradeable fashion, by anticipating entry-points in our models where new capabilities can be attached. The model will keep a clear separation between the concepts related to the structure of a test (i.e. the organization of a test) and those related to the behavior inside the structure (i.e. the instructions). An important organizing principle will be to distribute the available instructions across the hierarchy of ICD elements accessed by the test components. Behavioral models will be offered as high-level imperative code with a formally defined grammar.

We are leaning toward the use of the Eclipse Modeling Framework (with the Ecore specialized modeling language) for the formalization of the different concepts of interest. This will allow us to have access to a number of existing tools to produce specialized editors, checkers and code generators. Test engineers will then have a rich environment to define their own test models based on the meta-model.

In conclusion, our proposal is to abstract away from the existing proprietary implementation solutions and to work at a shared design level. For this, mature model engineering techniques exist and can be used. An even more ambitious approach would be to share high-level test specifications, and to automatically support all the design and code production chain. This has been recently presented as one of the hot challenges in the field of avionics [19]. Our work will clearly not solve this problem, but can be considered as an intermediate step along the road.
Bibliography


