# THÈSE

En vue de l'obtention du

**DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE**

**Délivré par :**

Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)

---

**Présentée et soutenue par :**
**Alexandru-Robert-Ciprian GUDUVAN**

**le** jeudi 18 avril 2013

**Titre :**

A Model-Driven Development of Tests for Avionics Embedded Systems
Une approche dirigée par les modèles pour le développement de tests pour systèmes avioniques embarqués

---

**École doctorale et discipline ou spécialité :**
ED MITT : Sureté de logiciel et calcul de haute performance

**Unité de recherche :**
Laboratoire d'Analyse et d'Architecture des Systèmes (LAAS-CNRS)

**Directeur(s) de Thèse :**

Mme Hélène WAESELYNCK (Directrice de thèse)
Mme Virginie WIELS (Co-Directrice de thèse)

**Jury :**

M. Fabrice BOUQUET (Rapporteur)
M. Benoît COMBEMALE (Examinateur)
M. Yann FUSERO (Examinateur)
M. Yves LE TRAON (Rapporteur)
Mme Hélène WAESELYNCK (Directrice de thèse)
Mme Virginie WIELS (Co-Directrice de thèse)

Day of the defence: the $18^{th}$ of April, 2013

# Abstract

The development of tests for avionics systems involves a multiplicity of in-house test languages, with no standard emerging. Test solution providers have to accommodate the habits of different clients, while the exchange of tests between aircraft manufacturers and their equipment/system providers is hindered. We propose a model-driven approach to tackle these problems: test models would be developed and maintained in place of code, with model-to-code transformations towards target test languages. This thesis presents three contributions in this direction. The first one consists in the analysis of four proprietary test languages currently deployed. It allowed us to identify the domain-specific concepts, best practices, as well as pitfalls to avoid. The second contribution is the definition of a meta-model in EMF Ecore that integrates all identified concepts and their relations. The meta-model is the basis for building test model editors and code generation templates. Our third contribution is a demonstrator of how these technologies are used for test development. It includes customizable graphical and textual editors for test models, together with template-based transformations towards a test language executable on top of a real test platform.

I dedicate my dissertation work to my family, my friends and to Calopsitta.

Dedic această teză familiei mele, prietenilor şi Calopsittei.

# Acknowledgements

It is my pleasure to acknowledge here those people whose presence in my life contributed to this dissertation : my academic and industrial advisors, colleagues, friends and family.

I am truly indebted to Hélène Waeselynck, my academic advisor, for her mentorship. You taught me not only the rigours and intricacies of scientific research, but also how to be a better person. Your humane openness, support and encouragement during my times of doubt were invaluable. This thesis would not have been possible without your wisdom and guidance. **Thank you!**

I also owe sincere thankfulness to Virginie Wiels, my academic co-advisor. Thank you for your implication in my doctorate, your important suggestions and remarks, as well as for the confidence you have shown in me and for your kindness.

I would like to express my gratitude to Yann Fusero, my industrial advisor, for his sustained support, as well as for having taught me the rigours of the industrial approach.

I would like to express my sincere appreciation to Michel Schieber, my industrial co-advisor, for his interest and participation in my doctorate, as well as for having introduced me to a wide range of fields and activities.

Special thanks go to Guy Durrieu for our valuable discussions and his caring and concern regarding this project.

I would like to thank Yves le Traon and Fabrice Bouquet for having accepted to review my thesis, as well as Benoît Combemale for having accepted to be a member of my defence committee.

I would like to express my gratitude to the management of Cassidian Test & Services: Philippe Lasman and Philippe Quichaud, as well as to the management of the LAAS-CNRS (Laboratory for the Analysis and Architecture of Systems): Karama Kanoun, Mohamed Kaaniche, Raja Chatila, Jean-Louis Sanchez and Jean Arlat, for having welcomed me.

It has been a pleasure to work with my colleagues at Cassidian Test & Services. Special thanks go to Florent Paitrault for our rich discussions and technical support, as well as to Gilles Ballanger, Guilhem Bonnafous, Mathieu Garcia and Etienne Allogo, for our rich interactions and especially for their implication in the development of our prototype. I am thankful to Valérie Aouba for her help, especially towards the end of my doctorate, as well as for having taken the time to come to my thesis defence. I am highly indebted to Caroline Plana-Robert for her kindness and support, as well as for giving me the right advice at the right time.

I would also like to thank my colleagues at the LAAS-CNRS, particularly Pierre André, Maxime Lastera and Fernand Lone-Sang.

I am also grateful to Emmanuelle Beatse and Françoise Ducor at Cassidian Test & Services, and Sonia de Sousa at the LAAS-CNRS, for all their help.

I would like to express my deep appreciation to the people I worked with within the context of the VISTAS (Virtually Integrated System Tests between Airframers and Suppliers) project. I would like to express my gratitude to Gérard Montariol for his tutorship. Special thanks go to Olivier Fourcade, Noëlle Marchal and François Ruiz, Dominique Dubost and Jean-Michel Lecuna, Mossad Khimoun and Christophe Ginestet, for our fruitful collaboration.

Special thanks go to Albert Lahana for having accepted me as his student and for having taught me so much about myself through music, as well as to Céline Romero for having coached me with her kind and wise words.

I am highly indebted and thoroughly grateful to all my friends. My dissertation would not have been possible without your constant motivation, support and love. I would like to thank Roxana Albu, Ioana-Mihaela Geantă and Miruna Stoicescu, whom while pursuing their own doctorate have been extremely close to me, emotionally and professionally. I would also like to thank Julien Eballard, Alexandra-Mihaela Niculae, Iuniana Oprescu, Alexandra Pencea and Diana Stoica for our enduring friendship. You are all like a second family to me! I would also like to express my gratitude to Jérome Haquin. Special thanks go to Benoît Baccot, David Brown, Vivian Madesclaire and François-Henry Rouet, as well as to Mihaela-Elena Bâclea, Daniel Ionescu, George Ionescu and Silvia-Maria Mihalea, for their friendship and for the quality time spent together.

I would like to thank Calopsitta for her companionship.

I am grateful to Paula Begoun for having given me great skin for the day of the defence.

Last, but not the least, I would like to thank my family. Without the love and support of my parents, Tanţa and Qwo-Wadis Guduvan, and of my grandparents, Tinca and Marin Maxim, and Elisabeta Kedves and Alexandru Guduvan, I would not have come this far. I would also like to thank Areta-Liliana Guduvan for her constant support. Special thanks go to Emilia and Dumitru Gudovan for their kindness.

# Contents

# List of Figures

# List of Tables

# Listings

# Glossary

| | |
|---|---|
| *ADIRS* | Air Data Inertial Reference System |
| *AFDX* | Avionics Full-Duplex Switched Ethernet |
| *ARINC* | Aeronautical Radio, Incorporated |
| *ASN*.1 | Abstract Syntax Notation One |
| *ATL* | Automatic Test Language (internal to Cassidian Test & Services) |
| *ATLAS* | Automatic Test Language for All Systems |
| *ATML* | Automatic Test Mark-up Language |
| *BDD* | Binary Decision Diagram |
| *BNF* | Backus-Naur Form |
| *CAN* | Controller Area Network |
| *CBCTC* | CycleByCycleTestComponent |
| *DSL* | Domain-Specific Language |
| *EMF* | Eclipse Modeling Framework |
| *ETSI* | European Telecommunications Standards Institute |
| *FSM* | Finite State Machines |
| *FWS* | Flight Warning System |
| *GMF* | Graphical Modeling Framework |
| *GSM* | Global System for Mobile Communications |
| *GUI* | Graphical User Interface |
| *HiL* | Hardware-in-the-Loop |
| *HSPL* | High-Level Scripting Programming Language |
| *ICD* | Interface Control Document |
| *IDE* | Integrated Development Environment |
| *IMA* | Integrated Modular Avionics |
| *IP* | Internet Protocol |
| *ITU* | International Telecommunication Union |
| *LCS* | Launch Control System |
| *LTS* | Labelled Transition Systems |
| *M2M* | Model to Model |
| *M2T* | Model to Text |
| *MAC* | Media Access Control |
| *MaTeLo* | Markov Test Logic |
| *MATLAB* | Matrix Laboratory |
| *MC/DC* | Modified Condition/Decision Coverage |
| *MCUM* | Markov Chain Usage Model |
| *MDE* | Model-Driven Engineering |
| *MiL* | Model-in-the-Loop |
| *MMI* | Man Machine Interface |
| *MOF* | Meta Object Facility |
| *MTC* | Main Test Component |
| *MTL* | Model to Text Language |
| *NASA* | National Aeronautics and Space Administration |
| *OCL* | Object Constraint Language |
| *OMG* | Object Management Group |
| *OOPL* | Object-Oriented Programming Language |
| *OSI* | Open Systems Interconnection |
| *PL* | Proprietary Language |
| *PLTL* | Propositional Linear Temporal Logic |
| *PTC* | PeriodicTestComponent |
| *SCARLETT* | SCAlable & ReconfigurabLe Electronics plaTforms and Tools |
| *SDL* | Specification and Description Language |
| *SiL* | Software-in-the-Loop |
| *SMV* | Symbolic Model Verifier |
| *STC* | SequentialTestComponent |

## GLOSSARY

| | |
|---|---|
| $STELAE$ | Systems TEst LAnguage Environment |
| $SUT$ | System under Test |
| $TM$ | TestMonitor |
| $TTCN-3$ | Testing and Test Control Notation Version 3 |
| $UC$ | User Code |
| $UIO$ | Unique Input Output |
| $UML$ | Unified Modeling Language |
| $UMTS$ | Universal Mobile Telecommunications System |
| $UTP$ | UML Testing Profile |
| $VDM-SL$ | Vienna Development Method Specification Language |
| $VL$ | Virtual Link |
| $WCET$ | Worst-Case Execution Time |
| $XMI$ | XML Metadata Interchange |
| $XML$ | Extensible Markup Language |

# 1 Introduction

This doctoral thesis was a collaboration between three organisations:

- an industrial partner: Cassidian Test & Services (an EADS company) of Colomiers, France,
- and two academic partners - research laboratories of Toulouse, France:
  - the Laboratoire d'Analyse et d'Architecture de Systèmes (LAAS-CNRS),
  - the Office National d'Études et de Recherches Aérospatiales (ONERA).

The industrial partner is a provider of test execution automation solutions for avionics embedded components and systems, covering integration testing as well as hardware testing at the production and maintenance life-cycle phases.

Testing is one of the main verification and validation means, aiming for the removal of faults [1]. In the field of avionics, testing is a crucial step needed for the certification of the systems that are installed inside an aircraft [2]. The automation of test execution is a long-time subject that is as important today as ever because of the major benefits it offers when compared to manual testing: the long-term effort associated with the testing activity is decreased, the execution of test cases is reproducible and can be performed with limited or no human supervision.

The industrial context of our work is the in-the-loop integration testing of avionics embedded systems. We were motivated by the observation that existing solutions for test execution automation in our industrial context no longer respond to stakeholder needs. The world of test languages, in which test cases are developed, comprises numerous proprietary test languages and is heterogeneous: each test language offers different functionalities and common functionalities are offered in different manners. In some cases, a test language offers a same functionality in more than one manner. As no standard is emerging in this field and existing test languages are different, the exchange of test artefacts (e.g., test cases and data) between stakeholders is hindered (e.g., between aircraft manufacturers and equipment/system providers). This situation is also challenging for test solution providers who have to cater to the individual habits of each client. Moreover, test engineers do not benefit from the latest methodologies and technologies issued from software engineering to their full extent. This could prove useful for increasing test engineer productivity and test case quality, while facilitating the maintenance and customization of the test development environments.

In order to address all these needs, we propose the application of model-driven engineering methodologies and technologies, issued from software engineering, in the field of test implementation for avionics embedded systems. High-level, platform-independent test models would replace current practice, where test code occupies the central position. For their execution, test models would be translated into existing executable test languages. This proposal is motivated by the fact that test cases are pieces of software and consequently can and should benefit from the latest software development methodologies and technologies.

Chapter 2 proposes a state-of-the-art presentation of the field of testing. We discuss existing work focusing on two major issues: test design (test selection and test oracle) and test implementation (test development methodologies and formalisms). Although most existing academic work addresses the design issues, test implementation ones are also very important from an industrial viewpoint.

Chapter 3 discusses the specificities of our industrial context. We present the life-cycle, behaviour and interfaces of avionics embedded systems, the associated test platforms and the evolving needs expressed by stakeholders in this field, that motivated our work.

Our first contribution (Chapter 4) consists in the analysis of a sample of test languages currently used in the industry. To the best of our knowledge, this is the first time such an analysis is performed. We targeted a sample of four proprietary test languages currently used in real-life, together with two test languages used in the automotive and respectively telecommunications and distributed systems domains. The latter two were chosen for comparison purposes. This analysis allowed us to identify the set of domain-specific features/concepts, best practices, as well as pitfalls to avoid. Secondly, this study investigated the similarities and differences between the test languages that were analysed. A too great heterogeneity could have impeached any homogenisation attempt, while a lower heterogeneity could have helped us identify the "best"test language is our sample and try to improve it. We found that test languages in this field are sufficiently similar in terms of offered functionalities, but that none of them seems an adequate candidate for future improvement. This motivated us to propose a new model-driven solution to replace existing practice.

Our second contribution (Chapter 5) consists in the definition of a test meta-model that integrates the rich set of domain-specific concepts in a homogeneous manner, while taking into account the best practices in the field. We used meta-modelling as an instrument in order to define our domain-specific concepts and their relations. Moreover, it provided us access to a rich set of existing free and open-source tools: graphical and textual model editors, checkers and automatic code generators. This enabled us to define a mixed (graphical and textual) customizable test model development environment.

The third contribution (Chapter 6) consists in the implementation of a first prototype on top of a real integration test platform: the commercially available U-TEST Real-Time System [3] developed at Cassidian Test & Services. We used a template-based model-to-text transformation, targeting an executable proprietary test language. The target test language did not exist at the time of our first analysis of the field and consequently provided us the opportunity to challenge the genericity of our test meta-model. We proposed a method for organising the automatic code generation templates and the automatically generated code and files. In order to simplify the implementation task, we based this organisation on the test meta-model architecture. Moreover, this prototype allowed us to evaluate the employed technology with regard to possible future industrialization. The model-to-text approach corresponded to our needs. We did not require more complex model transformation functionalities. The definition of automatic code generation templates was straightforward, with the automatic code generation step for a given test model being quasi-instantaneous.

With the prototype, we succeeded in demonstrating a number of test engineer activities on two realistic use-cases of simple and medium complexity (Chapter 7): test model development, implementation and execution.

Chapter 8, comprising the conclusion and perspectives of our work, ends this dissertation.

# 2

# State of the Art - Testing

This chapter presents the state of the art on the subject of **testing**, together with the main issues that are encountered and for which solutions have been or are currently being researched.

Testing is a dynamic verification method aiming at fault removal [1]. It consists in the stimulation of the system under test (SUT) with valued inputs and the analysis of its outputs with respect to the expected behaviour. Testing is one of the four major classes of verification techniques. The remaining three are: static analysis [4, 5, 6], model verification/checking [7, 8] and theorem proving [9, 10]. Although there is cross-fertilisation between these different techniques [11], they are most often used independently one from the other. The chosen technique depends on the life-cycle phase where the verification is performed, as well as the type of verification problem.

The testing process uses test cases that are submitted to the system. A test case is "*a set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement*" (from [12]).

In the field of testing, two main challenges are usually addressed: **test design** and **test implementation**. Test design concerns the definition of the logic of a test, while test implementation deals with the development of test code for execution automation purposes on top of given test platforms. These two issues can be addressed at different integration levels of the tested artefact: unit, sub-system, system and so forth.

Let us now look in more detail at the two test design and implementation challenges.

Test design can be split into two issues: **test selection** (Section 2.1) and **test oracle** (Section 2.2). Test selection concerns the specification of test behaviour and data that are pertinent with regard to the required verification objective. The test oracle deals with the manner in which a verdict can be taken regarding the conformity between the observed behaviour exhibited by the system under test and the expected behaviour.

Test implementation deals with the problems of **test development methodologies** (Section 2.3) and **test development formalisms** (Section 2.4). Test development methodologies cover guidelines that test engineers must follow when developing tests. Test development formalisms cover technologies that allow the definition of tests that are automatically executable.

Most existing academic work focuses on test design, with test selection being a topic of research more commonly tackled than the test oracle. Although the number of published research on test implementation is somewhat limited, it nevertheless represents important challenges. The implementation of test cases, in order to automate their execution, is a major issue in an industrial context. Regression testing benefits directly from this. Test implementation decreases the effort associated with the testing activity. The number of test cases that can be executed in a limited time is increased. Tests can be executed when the test platforms and SUT are idle (such as during night-time), without or with limited human supervision. Automated tests can have a reproducible execution, which is not achievable when they are entered manually by a test engineer/operator. All of these benefits justify the importance of test automation and the motivation for our work, as will be also mentioned in the conclusion of this chapter.

## 2.1 Test Selection

With the exception of trivial cases, exhaustive testing of all the possible behaviours that a system can exhibit - with all possible input data, is not achievable. Consequently, a small set of test cases that cover a subset of the input data domain must be identified. In order to select the relevant test cases, criteria that guide the selection are defined, covering different aspects of the system under test. Some criteria are linked to a model of the structure of the system or to a model of the functions that the system offers. In software testing, these two approaches are called

- **Structural Testing** (Subsection 2.1.1),
- **Functional Testing** (Subsection 2.1.2).

Unfortunately, the imperfection of the different test selection criteria to identify specification and implementation faults constitutes a major limitation of these approaches. This observation has motivated work on:

- **Probabilistic Generation** [13, 14, 15, 16] of input data, based on test criteria, in order to allow the multiple activation of system behaviours with different valued inputs, for increased fault identification. This is different from probabilistic usage profiles that help identify most-used behaviours of the system and concentrate testing on them (discussed at the end of Subsection 2.1.2),

- **Mutation Analysis** [17, 18, 19]) for the evaluation of the fault-revealing power of a set of test cases. A given system is applied mutation transformations and existing tests are evaluated with regard to their capacity to detect the inserted faults.

For a high-level overview of test selection, including information on its industrialisation, please refer to [20, 21, 22, 23].

### 2.1.1 Structural Testing

Structural testing "*takes into account the internal mechanism of a system or component*"(from [12]). The set of criteria that govern the selection of test cases in structural testing are defined following the structural analysis of the software programs. A white-box view of the software is considered. The model of the software that is used by structural testing approaches is the control graph [24], possibly enriched with information on the data flow [25].

The control graph, constructed from the source code of the software program, offers a compact view of the software structure. It contains one entry node and several potential exit nodes. The nodes of this graph comprise instructions that are executed sequentially. The arcs between nodes define possible transfers of control between the nodes. An execution of the software can thus be considered as a complete path between the entry node and one of the exit nodes, the path that was taken being determined by input data produced by the test case.

A stringent selection criterion consists in the activation, at least once, of each executable path between the entry node and the exit nodes (the "all paths"criterion). A path is executable if input data that activates it exists.

In reality, this criterion is difficult to use. Firstly, the problem of non-executable paths identification is undecidable in the general case. Secondly, whenever a software contains a loop, the number of executable paths greatly increases and can even be infinite. In such cases, one is forced to accept less stringent criteria, such as "all instructions"or "all branches". These two criteria demand that all the nodes and respectively all the arcs are activated at least once during testing. A refinement of branch testing is the Modified Condition/Decision Coverage (MC/DC) criterion [26, 27], used to cover the logic of branch predicates.

The control graph can be enriched with information relative to the manipulation of variables inside the software. Data flow criteria consequently propose the coverage of sub-paths between the graph nodes containing the variable definitions (a value is assigned) and the nodes or arcs using these values in the computation of mathematical or logical expressions. Different types of specialized criteria exist, such as "all variable definitions"or "all uses".

In general, structural testing is used for software components of reduced size, as the control graph rapidly increases in complexity with the increase in size of the targeted source code.

### 2.1.2 Functional Testing

Functional testing "*ignores the internal mechanism of a system or component and focuses solely on the outputs generated in response to selected inputs and execution conditions*"(from [12]). It corresponds to a black-box view of the system under test. This approach depends on a model that describes the expected software behaviour in terms of inputs and outputs. In comparison with the structural testing approach, there is no standard model (such as the control graph) that can be used. Consequently, functional testing covers a wider range of methods and techniques, which are dependant on the formalism used for modelling the behaviour of the system. We discuss here some well-known approaches:

- **Input Domain Characterisation**,
- **State-Based Approach**,
- **Algebraic Specification**,
- **Decision Table**,
- **Probabilistic Usage Profile**.

**Input Domain Characterisation** This technique deals with the modelling of the input domain using a partitioning approach. The input domain is divided into a finite number of sub-domains, or equivalence classes, separating valid and invalid value sets. The equivalence classes are deduced from the analysis of the system specification and its input data types. The classes group values that should activate similar software program functionalities. The criterion consists in choosing an element from each equivalence class. This approach can be enriched with the consideration for values at the boundaries between sub-domains. Note that the domain characterization is most-often multi-dimensional, as in Category-Partition Testing [28], where each category represents an input characteristic to be partitioned. Various strategies have been proposed to combine the values for the orthogonal characteristics [29].

**State-Based Approach** This technique consists in the modelling of the states that a software program can have, together with the operations or events that lead to changing states. The modelling of the system is achieved either by using set theoretic or state-transition formalisms.

Languages issued from set theory are used for functional testing, yielding approaches for the Vienna Development Method Specification Language (VDM-LS) [30], Z [31] and B [32, 33, 34]. The associated methods contain two steps. First, test cases are selected for each operation that is analysed individually. Criteria for operation analysis deal with the coverage of their before/after predicates. Next, operation call sequences are constructed, where each operation in the sequence sets the software program in a specific state in which the following operation can be activated.

Two basic versions of state-transition models used for testing are: Finite State Machines (FSM) and Labelled Transition Systems (LTS). We discuss both of these techniques next.

A FSM is a graph that has a number of nodes, with each node representing a state of the system, while the arcs between the nodes represent transitions between system states. A finite alphabet of input and output events allow the annotation of each transition with the input that produces it and with the output it produces after it has been triggered. The test selection criteria defined for a FSM are based on the coverage of the graph [35], such as: the coverage of all states, the coverage of all transitions [36, 37], the coverage of all transition pairs [38]. More sophisticated versions of transition testing have been defined. They target exhaustiveness with respect to a simple fault model, inducing transfer (the transition leads to a state different from the one that is specified) or output (the transition does not generate the expected output) errors. The detection of transfer errors leads to the need to identify the arrival state. As this state is not always directly observable, it must be identified by looking at one or more sequences of events to which it responds in a manner different than any other state. Several solutions exist for the generation of such sequences of events: the W method [39], Distinction and Unique Input Output (UIO) sequences [40, 41]. These methods have also been looked into in the case of indeterministic machines [42].

The LTS-based technique is usually employed for testing communicating systems [43, 44]. This formalism is usually not directly used in practice, with the specification being defined in a higher-level language with its semantics defined in LTS, such as the Specification and Description Language (SDL). The generation of tests can be performed by linking the LTS with the test objectives that specify the sequences of events to be tested. The test objectives can be determined in an ad-hoc manner or can be automatically generated from the specification (e.g., objectives related to the coverage of the source SDL model transitions).

Test methods have also been investigated for timed [45, 46, 47] and data [48] extensions of the state-transition formalisms.

**Algebraic Specification** An algebraic specification offers an axiom-based description of the functionalities of a software program. Testing an axiom consists in the instantiation of its variables and the verification that the software program satisfies the formula that is obtained. The manner in which an axiom is instantiated is based on selection hypotheses [49, 50]: the uniformity hypotheses define input homogeneous subdomains (i.e., equivalence classes) based on their capacity to reveal faults, regularity hypotheses limit the size of the formulas that can be constructed by axiom unfolding. This approach has been extended to the analysis of LUSTRE descriptions [51] and process algebras [52]. Recent work has proposed a way to encode the test selection problem and hypotheses in a formal test generation environment based on the theorem-prover Isabelle [53].

**Decision Table** A decision table allows the identification of the input combinations that influence the behaviour of the system under test and can be considered a high-level language for representing combinational functions (without memory). A decision table contains two parts: a list of conditions or predicates on the inputs and a list of actions to be performed (outputs). Each table column defines a rule that links the combination of condition evaluation values to a list of expected actions. The criterion associated with the table covers the activation of each rule at least once.

**Probabilistic Usage Profile** The basic principle behind statistical usage testing is that the tests are based on the anticipated operational usage profile of the system [54, 55]. This technique is consequently based on the modelling of the manner in which a user can interact with the system. A usage profile is for example a Markov Chain Usage Model (MCUM) [56] containing nodes and arcs. Each node is a state of the system, with arcs being interactions from the user that change the system state. Each arc is assigned a probability that corresponds to the chance that a user performs that respective interaction. The MCUM is usually derived from a higher-level specification. Standard Markov techniques can be used to analyse the usage profile. An example of a generic tool using this technique is Markov Test Logic (MaTeLo) [57, 58], which allows automatic test case synthesis and code generation towards the TTCN-3 [59] test language. Others have applied usage profiles for the testing of military simulation systems [60], mobile networks [61] and web applications [62]. In [63] a richer extended operational usage profile is proposed, that actually consists of three profiles (the process, structural and data profiles) taking into account different aspects of the system under test.

## 2.2 Test Oracle

The automatic analysis of the test results is desirable and becomes necessary when a great number of input/output data of the system under test have been selected. This is where the formalisation of a test oracle comes in handy.

Solutions that offer the best results are based on the existence of a formal specification of the target software program. This specification is usable for determining the expected results. In Subsection 2.1.2 we have presented some examples of approaches based on formal specifications, which are used in functional testing.

Back-to-back testing represents another solution for the test oracle issue, which is especially useful for systems that have software diversification for fault avoidance purposes. This solution consists in the comparison of the output data produced by the software variants. In case the variants produce divergent results, faults are revealed. Back-to-back testing remains nevertheless an imperfect solution, because of the potential presence of correlated faults that lead the variants to produce the same incorrect results.

Other partial test oracle solutions can be employed, on a case-by-case basis, by using verisimilitude checks on the test results: the verification of the coherence between different data, the verification that data is bounded, and so forth. More sophisticated checks use invariants that can also take temporal aspects into account [64, 65]. Executable assertions/contracts can also be included into the software program code [66, 67, 68], allowing fine-grained verification to be performed. The assertions allow the detection of erroneous internal system states, as well as the detection of the violation of pre/post-conditions or operations.

## 2.3 Test Development Methodologies

Test development methodologies cover guidelines that test engineers use when they develop tests, allowing them to define high quality test code (e.g., annotated with comments, structured, understandable, documented, reusable). In the field of software testing, the book [69] offers an extensive overview of existing practices, as well as real-life case studies.

The book discusses scripting techniques that can guide test development from the perspective of programming techniques. Five scripting techniques are presented, each one with its strengths and weaknesses: linear, structured, shared, data-driven and keyword-driven scripting. A linear script is a simple set of sequential instructions that stimulate the system under test and verify its outputs. A structured script benefits from execution flow control instructions, being similar to a structured program. Shared scripts are scripts that can be reused across several test cases. Data-driven scripts separate the behaviour of the test case (e.g., stimulate a text input) from the input data (e.g., a specific string) that is used. Finally, keyword-driven scripts are enhanced versions of data-driven ones, where some part of the logic of the test case is transferred to the input data.

The comparison of system under test output with expected output (i.e., the test oracle issue) is discussed as well. Comparators are pieces of software that deal with such issues. They can perform their task dynamically, during the execution of the test case, or in post-processing. They can have different levels of sensitivity, looking for identical of more logical correspondences between observed and expected outputs.

Finally, [69] looks into issues concerning test managers/directors, test organization, test management, as well as metrics for measuring test quality.

These methodologies are most of the time informal, being found in text documents that test engineers are supposed to read and apply. A manner in which the correct application of these methodologies can be more strongly enforced is by introducing them in the test development formalism that a test engineer uses. As such, the test engineer is constrained to respect the methodology, in the same manner in which a programmer must respect the grammar of a programming language when developing a software program.

## 2.4 Test Development Formalisms and Associated Tools

Test development formalisms and tools are technologies that allow the production of test cases that are automatically executable on a given test platform. Three major types of solutions have been identified: **capture & replay tools**, **test frameworks**, **test languages** and **test modelling**. We discuss each one of those next, with a couple of examples illustrating each solution type.

**Capture & Replay Tools** A capture and replay tool is a test tool that records test inputs as they are sent to the software under test during manual testing. The input cases stored can then be used to reproduce the test at a later time. These types of tools are most commonly applied to Graphical User Interface (GUI) testing. This approach usually leads to fragile tests [69]. We present here some tools.

Borland SilkTest [70] is a tool for automated function and regression testing of enterprise applications. SilkTest uses an object-oriented representation of the software program, by identifying all windows and controls of the application under test as objects and defines all of the properties and attributes of each window. It can also identify mouse movement along with keystrokes. SilkTest can use both record and playback or descriptive programming methods to capture the dialogue boxes. SilkTest test scripts are stored in a proprietary test language called 4Test. It is an object-oriented language similar to C++, using the concepts of classes, objects, and inheritance. Pieces of code in Java, C++, C♯, as well as in Visual Basic .NET can be used for writing test scripts in SilkTest. This is especially useful when testing applications written in these languages. Other extensions supported by SilkTest cover testing for Internet Explorer (IE), Firefox and the SAP Windows GUI. SilkTest uses the Silk Bitmap Tool for the capturing and comparing of windows and application areas.

The Abbot Java GUI Test Framework [71, 72] is used for Java user interface (UI) testing. The framework contains two components: Abbot that offers the capacity to programmatically drive UI components, and the Costello scripting language that offers the capacity to launch, explore and control the application under test. In general, testing with Abbot consists of getting references (or handlers) to GUI components and either performing user actions on those components or making some assertions about their state. These operations may be done from either a high-level script written in Costello or directly from Java code (for example in a JUnit TestCase method). Costello scripts are saved in the Extensible Markup Language (XML) [73]. Scripts are primarily composed of actions, assertions, and component references. Actions are things that a user usually performs in order to operate a GUI component (e.g., clicking on a button, selecting a menu item or entering text). Component references represent actual GUI component instances within the application under test. Each has a unique identifier that may be used to indicate to other script steps the actual component on which one wishes to act. Finally, assertions allow the verification of the GUI state, for example by evaluating a predicate on the value of a GUI component property.

Selenium [74, 75, 76] is a portable software testing framework used in a different domain, that of web applications. Selenium provides a record/playback tool for test development. A test domain-specific language called Selenese is also provided. The tests can then be run against most modern web browsers, with Selenium being deployable on Windows, Linux, as well as Macintosh platforms. Recent work covers the mining of web application specifications from tests defined using Selenium [77].

**Test Frameworks** Test frameworks are commonly used in software engineering for unit testing. The xUnit generic term designates a framework for unit testing in a specific programming language, where the first letter is replaced depending on the programming language. All these variants are based on the approach proposed in SUnit, that was created for the Smalltalk programming language. Tests are written in the same programming language as the target application under test.

CUnit [78] is a lightweight system for writing, administering, and running unit tests in the C programming language. It offers developers basic testing functionalities with a flexible variety of user interfaces. CUnit is built as a static library linked with the application under test. It uses a simple framework for building test structures, and provides a rich set of assertions for testing common data types. Finally, several different interfaces are provided for running tests and reporting results. We mention here some variants, such as CppUnit [79], JUnit [80] and PyUnit [81].

**Test Languages** A test language is a domain-specific language (DSL) specifically created for the purpose of testing. We discuss some test language examples in order of their maturity, beginning with those that have been deployed in a production environment and ending with research prototypes.

Initially developed for the verification of distributed system and telecommunication protocol, the Testing and Test Control Notation Version 3 (TTCN-3) [82, 83] is a mature international standard evolving towards other fields, such as the testing of real-time [84, 85] and continuous [86] systems. TTCN-3 is a high-level test language that abstracts away from implementation details. It requires dedicated compilers or interpreters for execution. Nonetheless, it benefits from a standardised architecture for its implementation on top of test platforms [87]. Until its second version, the test language was specified as a set of tables and was called Tree and Tabular Combined Notation. Reading and editing this version of the test language required specific tabular editors. Beginning with its third version, TTCN was renamed. It now more closely reassembles a programming languages and can be edited with traditional textual editors. TTCN-3 was developed and standardized at the European Telecommunications Standards Institute (ETSI). This test language is widely used, mostly for the testing of telecommunication protocols, such as: Global System for Mobile Communications (GSM - 2G), Universal Mobile Telecommunications System (UMTS - 3G), Bluetooth. An interesting aspect of TTCN-3 is its use of the Abstract Syntax Notation One (ASN.1) [88] formalism for the description of the message types that are send to and received from the system under test. ASN.1 is an international standard of the ITU Telecommunication Standardization Sector (ITU-T).

Rational Test RealTime [89] is a tool developed and sold by IBM/Rational, targeting software component testing and the runtime analysis of embedded software. This tool integrates the ATTOL test language. A preprocessing module transforms ATTOL test scripts into a target programming language, which is usually the one in which the tested software application is written (e.g., the C programming language). One interesting aspect of this test language is that it allows foreign code (i.e., in a programming language different than ATTOL) to be embedded into its test scripts. Moreover, it offers short-cuts for the implementation of simple test strategies (e.g., interval values coverage), as well as simple oracles based on the comparison of resulting data with regard to expected data. The execution of the test scripts automatically generates a detailed rapport that can serve for testing process documentation.

The Test Specification Language (TSL) [90] is used for the definition of formal tests for software systems. TSL was developed by Siemens Corporate Research. This test language has been used to test commercial software in a production environment, such as the commands of a software management system and software for a process control system. TSL is based on the Category-Partition method [28] (see Subsection 2.1.2). This represents a systematic way of analysing a system's functional specification to produce a well-defined partition of each function's input domain and a description of each function's expected results. Executable test cases are automatically generated from the more abstract formal test specifications defined in TSL by combining values for the different partition elements.

A prototype language based on Python was developed at the National Aeronautics and Space Administration (NASA) [91], specifically for their Constellation rockets Launch Control System (LCS) project. It was used as a specialized monitor and control language for programming both tests and the control parts of some applications (checkout and launch processing for flight and ground systems). The language has been implemented as a library that extends the Python scripting language, and validated in a successful demonstration of capability required for the NASA Constellation Launch Control System (LCS). The design of the Domain-Specific Language (DSL) was inspired by the procedural programming paradigm, rather than by object-oriented programming, in spite of the fact that Python is object-oriented. The DSL focuses more on functionalities that on structures. In addition to the domain-specific constructs, all of Python's programming constructs are reused and made available to the programmer. Valuable feedback is offered in [91] on issues to be tackled when specifying a new test language (e.g., defining it from scratch or reusing an existing language), as well as feedback on the used technologies (e.g., Python, PyDev Python development environment for Eclipse).

**Test Modelling** Test modelling is based on the assumption that test cases are pieces of software, and as such can benefit from advanced software engineering methodologies and technologies, such as model-driven engineering (MDE) [92]. The introduction of MDE methodologies and technologies to the field of testing is an active field of research, although existing publications are not numerous. We do not refer here to model-based test selection techniques such as those already discussed in Subsection 2.1.2. Rather, we refer to test development.

Most existing work on test implementation solutions use the Unified Modeling Language (UML) [93] for the test models. Many projects have addressed the integration of the standardized UML Testing Profile (UTP) [94] and TTCN-3 [83]. The profile is used in [95] to produce TTCN-3 code (or code skeletons). A meta-model for TTCN-3 can be found in [96], later encapsulated within the TTworkbench platform [97]. A similar project at Motorola [98] uses the TAU tool suite [99].

Some authors proposed their own UML profiles. A UML profile and model transformations for web applications testing is discussed in [100]. In avionics, UML-based modelling of simulation software for model-in-the-loop testing is proposed in [101]. Also in avionics, [102] proposes test models conforming to a test meta-model (integrating automata-based formalisms), for the second generation of Integrated Modular Avionics (IMA) (SCAlable & ReconfigurabLe Electronics plaTforms and Tools (SCARLETT) project [103]). Previous work by the same authors includes automata-based test modelling for their RT-Tester platform [104, 105].

Meta-models for ITU-T languages such as TTCN-3 or SDL are investigated in [106]. The aim is to abstract away from concrete Backus–Naur Form (BNF) grammars and use a set of core concepts shared by a family of languages. Concepts identified by language experts are used as a support for building language meta-models.

Other artefacts employed during testing, in addition to the test cases themselves, can benefit from model-driven engineering. For example, SUT environment modelling is discussed in [107].

Similarly to this work, we focused on the usage of model-driven engineering methodologies and technologies for the implementation of tests. The industrial partner for this project, Cassidian Test & Services, is a provider of test execution automation tools and test platforms for avionics embedded components and systems.

## 2.5 Conclusion

In this chapter we presented the state of the art on testing. We discussed the major challenges in this field (test design and test implementation), together with the solutions that have been or that are currently researched.

Test design can be split into two issues: test selection and test oracle. Test selection is largely discussed in literature, in contrast to the test oracle. Solutions to the problem of relevant test selection include approaches such as: functional and structural testing, probabilistic generation of test input data and mutation analysis. Functional testing approaches are more numerous and diverse, as no single model is employed, in contrast to structural testing. The underlying models include input domain characterisation, state-based approaches, algebraic specification, decision tables and probabilistic usage profiles. Concerning the test oracle problem, automated solutions range from back-to-back techniques useful when redundancy is available, to techniques employing verisimilitude checks, complex invariants as well as executable assertions/contracts.

Test implementation can be split into two issues as well: test development methodologies and formalisms/tools. Test development methodologies guide test engineers in their coding activity, while test development formalisms and tools allow them to automate the coding (at least partly). In academia, these issues have received much less interest than test selection issues. Nevertheless, they are important in industrial environments, where test automation can provide numerous benefits that manual testing techniques can never achieve. Test development methodologies cover aspects ranging from the type of scripts that should be written to how the test oracle can be defined by comparing system under test outputs with expected results, as well as including the management of tests and the evaluation of their quality. The realm of test development formalisms and associated tools contains varied solution types: capture & replay tools mostly used for GUI testing, programming language-specific test frameworks used generally for unit testing, test languages, and more recent test modelling approaches that favour model-driven engineering methodologies and technologies instead of test implementation languages. Existing literature for test modelling is however limited, representing only a small fragment of the large body of work on model-driven engineering.

Our work is in the field of test development formalisms. We focused on the application of modern model-driven engineering methodologies and technologies in the field of avionics embedded systems testing. Chapter 3 presents the specificities of our industrial context, that justify and motivate the research direction proposed by this thesis.

# 3

# Industrial Context

This chapter briefly introduces our industrial context: the in-the-loop integration testing of avionics embedded systems. The analysis of the characteristics of our industrial context justifies the fact that a new model-driven approach would be useful, and consequently supports the relevance of the subject we chose to address in this document. We do not enter into too much detail, focusing only on information that is necessary for the understanding of the discussion of test languages that follows in the next chapter. Section 3.1 presents the specificities of an avionics embedded system. Section 3.2 discusses the architecture of a simplified test platform connected to the system under test. Section 3.3 overviews the evolving needs of the different actors in our industrial field. The fact that existing test development solutions no longer respond to these needs motivated our work. Section 3.4 concludes this chapter.

## 3.1 Avionics Embedded System

In this Section we discuss the specificities of avionics embedded systems: their life-cycle (Subsection 3.1.1), interfaces (Subsection 3.1.2) and behaviour (Subsection 3.1.3). We focus mostly on systems conforming to the Integrated Modular Avionics (IMA) architecture [2].

### 3.1.1 Life-Cycle and Testing Activities

The verification and validation of an avionics embedded system involves a rigorous process, with specific testing activities attached to the different life-cycle phases (for an overview, see for example [108]). Figure 3.1 provides a schematic view of the life-cycle of an avionics 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 usually does not need further functional validation, but hardware testing activities are still necessary to reveal manufacturing and ageing problems.

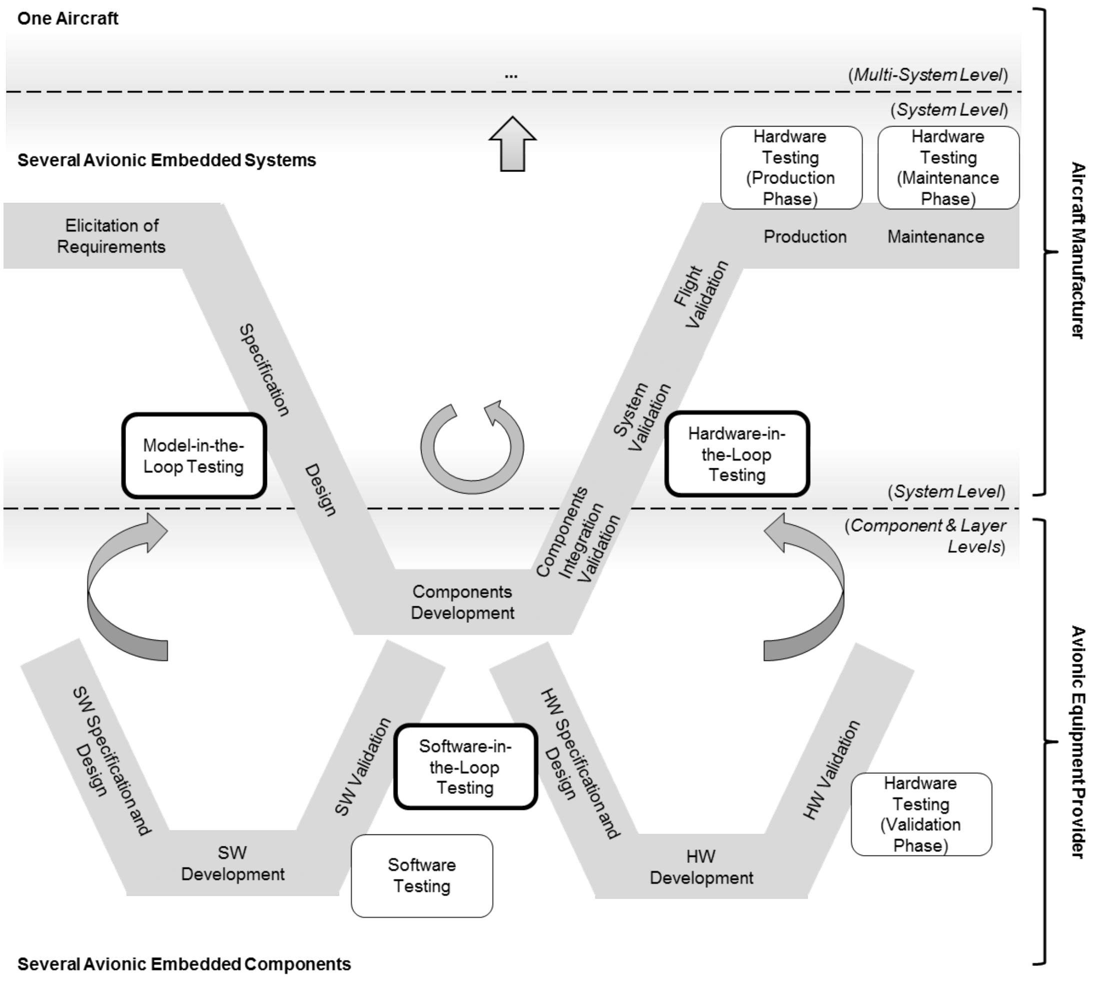

One Aircraft
...
(Multi-System Level)
(System Level)
Several Avionic Embedded Systems
Hardware Testing (Production Phase)
Hardware Testing (Maintenance Phase)
Elicitation of Requirements
Production
Maintenance
Specification
Flight Validation
System Validation
Model-in-the-Loop Testing
Hardware-in-the-Loop Testing
Design
(System Level)
Components Integration Validation
(Component & Layer Levels)
Components Development
SW Specification and Design
SW Validation
Software-in-the-Loop Testing
HW Specification and Design
HW Validation
Hardware Testing (Validation Phase)
SW Development
Software Testing
HW Development
Several Avionic Embedded Components
Aircraft Manufacturer
Avionic Equipment Provider

**Figure 3.1: System under Test Life-Cycle and Testing Activities -**

Our work focuses on in-the-loop testing phases that occur during the development process. We introduce their specificities below.

An avionics 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 3.1, in-the-loop testing comes in various forms:

- model-in-the-loop (MiL),
- software-in-the-loop (SiL),
- 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 artefact 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.

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 artefact 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.

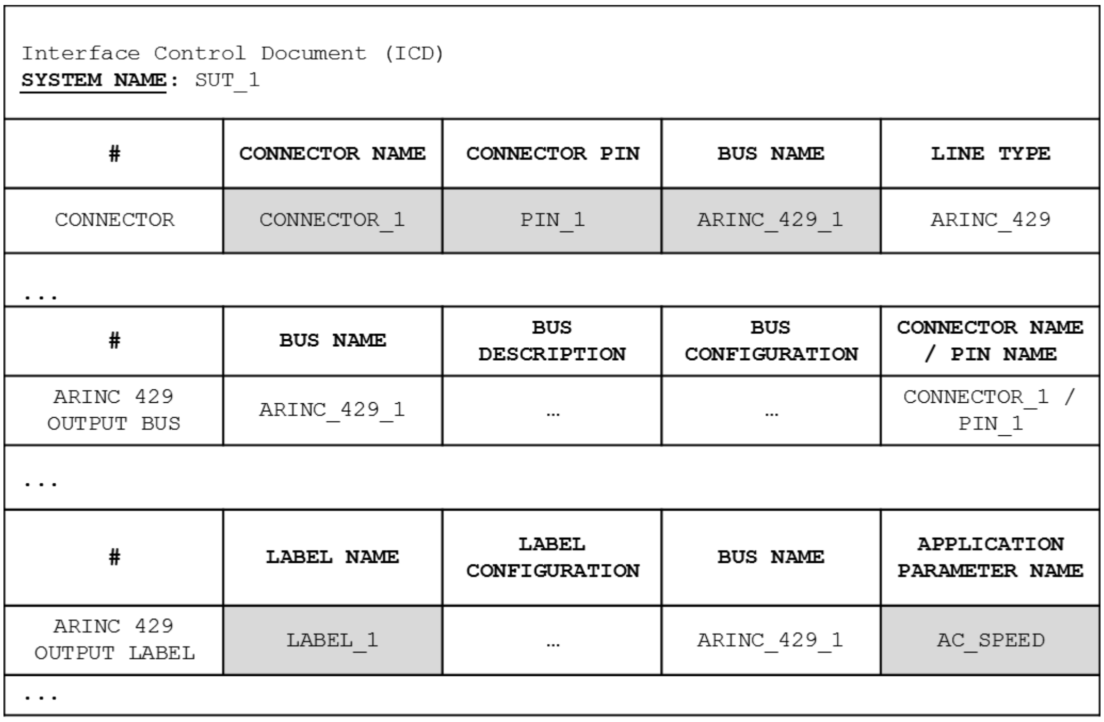
### 3.1.2 Interface Control Document (ICD)

The interfaces of an avionics embedded system are 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 3.2).

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 are attached to the physical connectors 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.

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) [109],
- ARINC 429 (Aeronautical Radio, Incorporated) [110],
- MIL-STD-1553B [111],
- and others.



Interface Control Document (ICD)
**SYSTEM NAME**: SUT_1

| # | CONNECTOR NAME | CONNECTOR PIN | BUS NAME | LINE TYPE |
|---|---|---|---|---|
| CONNECTOR | CONNECTOR_1 | PIN_1 | ARINC_429_1 | ARINC_429 |

...

| # | BUS NAME | BUS DESCRIPTION | BUS CONFIGURATION | CONNECTOR NAME / PIN NAME |
|---|---|---|---|---|
| ARINC 429 OUTPUT BUS | ARINC_429_1 | … | … | CONNECTOR_1 / PIN_1 |

...

| # | LABEL NAME | LABEL CONFIGURATION | BUS NAME | APPLICATION PARAMETER NAME |
|---|---|---|---|---|
| ARINC 429 OUTPUT LABEL | LABEL_1 | … | ARINC_429_1 | AC_SPEED |

...

**Figure 3.2: Interface Control Document Sample -**

For example, let us assume that an avionic embedded component possesses on its interface a connector conforming to the ARINC 429 standard. 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 3.2 shows what a corresponding ICD would look like. In real life an ICD is complex and large, containing sometimes tens of physical buses and thousands of application parameters.

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.

The ICD can contain additional information to that presented in the example, like the data type of the application parameter, 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 application parameter can be present. For example, such is the case when a component produces an application parameter that is consumed by several neighbouring components. Note that the lower-level interfaces can be of different types: AFDX or ARINC 429. Also, the component producing the parameter may be duplicated within the system for redundancy purposes.

Usually a tuple is used in order to uniquely identify a particular instance of an application parameter. In the above example of speed variable, a tuple could represent a path in the tree-like structure of the ICD:

“`SUT_1/ARINC_429_1/LABEL_1/AC_SPEED`”.

Some information in the path is redundant because of methodological constraints (e.g., the name of each application parameter on a bus is unique). Hence, the long identifier seen above can be shortened. A triplet usually suffices:

“`SUT_1/ARINC_429_1/AC_SPEED`”.

Even short identifiers yield long names for application parameters. In test procedures, it is often convenient to have aliasing mechanisms with symbolic names. The manner in which ICD elements are accessed by a test platform is discussed in Section 3.2.

### 3.1.3 Reactive Behaviour

Components of an avionics system typically exhibit a predominantly cyclic behaviour, where an execution cycle reads the input data and computes the output ones. For example, the value of the speed of the aircraft is periodically sent by the producing component to the consuming ones, 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.

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 actions include the modification 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.

## 3.2 Test Platform

A test platform for an avionics embedded system typically has the following components (Figure 3.3):

- a test controller,
- some test resources,
- a test network,
- a the test language,

The processing of the test logic is usually centralised, with the test controller being 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.

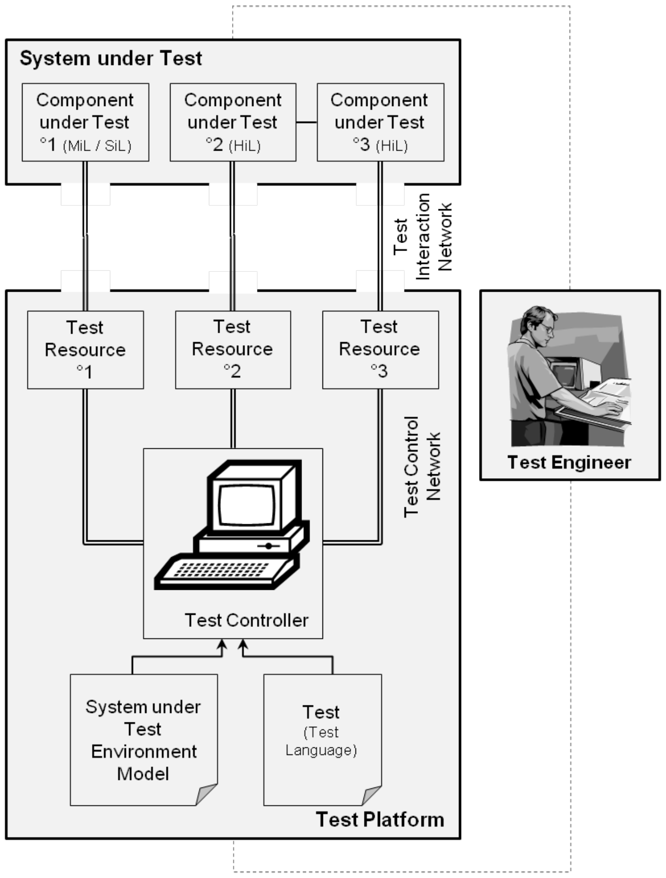


**Figure 3.3: Test Platform Architecture Overview -**

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 application parameters, 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 application parameters. 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.

## 3.3 Stakeholders and Evolving Needs

The in-the-loop integration testing of avionics systems involves the following types of stakeholders:

- test solution providers,
- test solutions users:
  - aircraft manufacturers,
  - and avionic equipment providers.

We had the opportunity to discuss and collect the needs expressed by representatives of all stakeholder types.

Historically, the aircraft manufacturer was in charge of all the integration activity for avionics 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 3.1, this developing business model (which is a consequence of the introduction of IMA architectures [2]) can be visualized as a tendency of the horizontal line delimiting the intervention of the providers 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 avionics system provider, opens an opportunity for collaboration between the two. A new type of interaction emerges, comprising the exchange of tests. Naturally, each actor has its own test languages, internal tools (e.g., test development environments) and test platforms that it uses for testing. Inherent incompatibilities between them severely limit the exchanges that can be done. In practice, the portability of tests from one environment to the other cannot be easily achieved. Portability needs also arise from the obsolescence management of test platforms: for example, existing tests should be reusable when changing a test resource on the test platform. To address these needs, a key feature of test languages is their ability to offer platform-independent abstractions, so as to hide the usage of test resources. Because of the multiplicity of proprietary test development tools, test solution providers have to accommodate the different habits of their customers. Customization is consequently demanded in order to allow the test solution provider to cater to these individual habits. Maintainability is useful for upgrading the test solution in a controlled manner, when demanded by a test solution user.

No standard test language exists or is emerging for the in-the-loop testing of avionics embedded systems. This contrasts other fields that use international standards, for example: the Abbreviated Test Language for All Systems (ATLAS) [112] and the Automatic Test Mark-up Language (ATML) [113] standards in hardware testing (production and maintenance life-cycle phases) or the Testing and Test Control Notation Version 3 (TTCN-3) [82] in the field of telecommunication protocols and distributed systems. These standardized solutions are not designed to address the specificities of our industrial context and as such are not directly reusable.

Other needs concern the capacity for the easy and rapid interpretation and writing of tests by test engineers. Readability is demanded in order to allow different stakeholders to understand one another's 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: the productivity of test engineers (i.e., number of lines of code by unit of time) and the quality of the test (i.e., reduction of programming errors). Writability is all the more important as test engineers are experts in avionics systems and not in general-purpose programming languages. This should be in favour of high-level, domain-specific, solutions.

All of these requirements motivated the investigation of test platform-independent, high-level, descriptions of tests with automated code generation facilities. Our work focuses on the introduction of model-driven engineering to the development of tests for in-the-loop testing of avionic embedded systems.

## 3.4 Conclusion

In this chapter we presented the industrial context of the doctoral thesis: the specificities of avionics embedded systems (life-cycle, interfaces and behaviour), their associated test platforms, together with the evolving needs expressed by stakeholders in this field: aircraft manufacturers, equipment/system providers and test solution providers.

This industrial context is complex:

- the SUT is verified at different maturity levels (model/software/hardware-in-the-loop),
- the SUT has a wide range of interface types at multiple hierarchical levels, organized in an ICD,
- a high number of stakeholders interact (aircraft manufacturers, equipment/system providers, test solution providers), each with its own habits and tools.

Existing test languages no longer respond to the complexity of the industrial context and to the needs expressed by the stakeholders. Moreover, test developers do not have access to the latest advances in software engineering, such as model-driven engineering. The transfer of these technologies from the software engineering domain to the field of test development for avionics embedded systems was investigated during this doctoral thesis.

Test models would thus become central to the test development activity, replacing current approaches where the test 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 and technologies [92], such as meta-modelling techniques and model-to-code transformations. The foundation of model-driven engineering is the meta-model, that constraints the definition of models in the same manner that a programming language grammar constraints the definition of code. In order to be able to define our own meta-model, specific to the in-the-loop testing of avionics embedded systems, we analysed a sample of test languages. It allowed us to identify the key domain-specific concepts and their relations, best practices, as well as problems that should be avoided. We present this analysis next, in Chapter 4.

# 4

# Test Languages Analysis

We present and analyse here a sample of six test languages. We chose four proprietary test languages used in our industrial context (in-the-loop testing of avionics embedded systems), together with two additional test languages issued from other fields. The latter two test languages were chosen for comparison purposes. This analysis allowed us to identify the set of domain-specific features that were integrated inside the test meta-model underlying our model-driven approach. Furthermore, this analysis also allowed us to identify best practices, as well as pitfalls to avoid.

Section 4.1 introduces our sample of six test languages and outlines our approach for their analysis.

Section 4.2 discusses generic features exhibited by the different test languages: whether they are compiled/interpreted, based on existing general-purpose programming languages or defined from scratch, and so forth.

Sections 4.3 to 4.6 discuss test-related features. We identified four broad categories:

- the organization of the tests (Section 4.3),
- the abstraction and access to the interfaces of the SUT (Section 4.4),
- the language instructions for performing test control and test interaction actions (Section 4.5),
- the management of time (Section 4.6).

Section 4.7 presents a synthesis of the principles that guided our meta-modelling work, based on the results of the test languages analysis.

Section 4.8 concludes this chapter.

## 4.1 Test Languages Sample

The sample of test languages includes:

- four proprietary languages from the avionics domain, which shall be named $PL_1$, $PL_2$, $PL_3$ and $PL_4$,
- TestML [114] from the automotive domain,
- Testing and Test Control Notation Version 3 (TTCN-3) [82, 83] from the networking and telecommunication domain.

The four proprietary test languages, from $PL_1$ to $PL_4$, are languages currently employed in the avionics industry. They have been chosen because they are representative of the practice among Cassidian Test & Services partners. The first one represents the offer of Cassidian Test & Services on the U-TEST Real-Time System [3] integration test platform. The identity of the three others cannot be disclosed. 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 chapter, 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 and TTCN-3. We did not include any of the test languages that were mentioned in Section 2.4, with the exception of TTCN-3, either because they are not related to our industrial context (e.g., targeting software unit testing) or because information concerning them is limited.

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 4.1. 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 its consideration was deemed relevant to us.

As shown in Table 4.1, 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 avionics embedded systems are also distributed systems, and their communication is based on interfaces conforming to a wide range

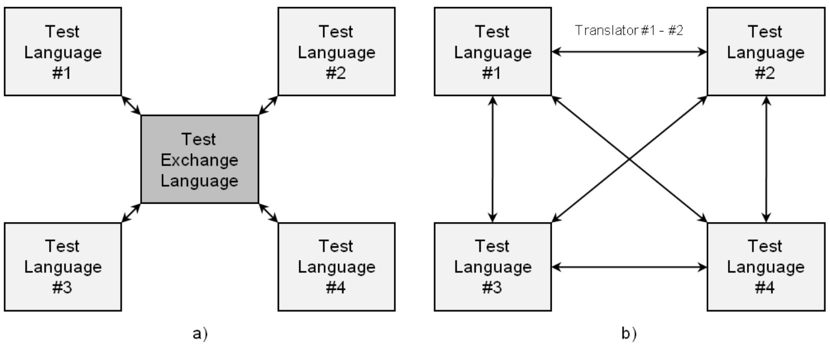


**Figure 4.1: Test Exchange Language -**

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.

The following types of data were available for the different test languages:

- test language documentation and user manuals ($PL_1$, $PL_3$ and $PL_4$), private presentations ($PL_2$),
- published articles and books (TestML and TTCN-3),
- test procedures written in test languages ($PL_1$ and $PL_3$),
- associated dedicated test development environments ($PL_1$ and $PL_4$), as well as test platform-specific configuration files ($PL_1$).

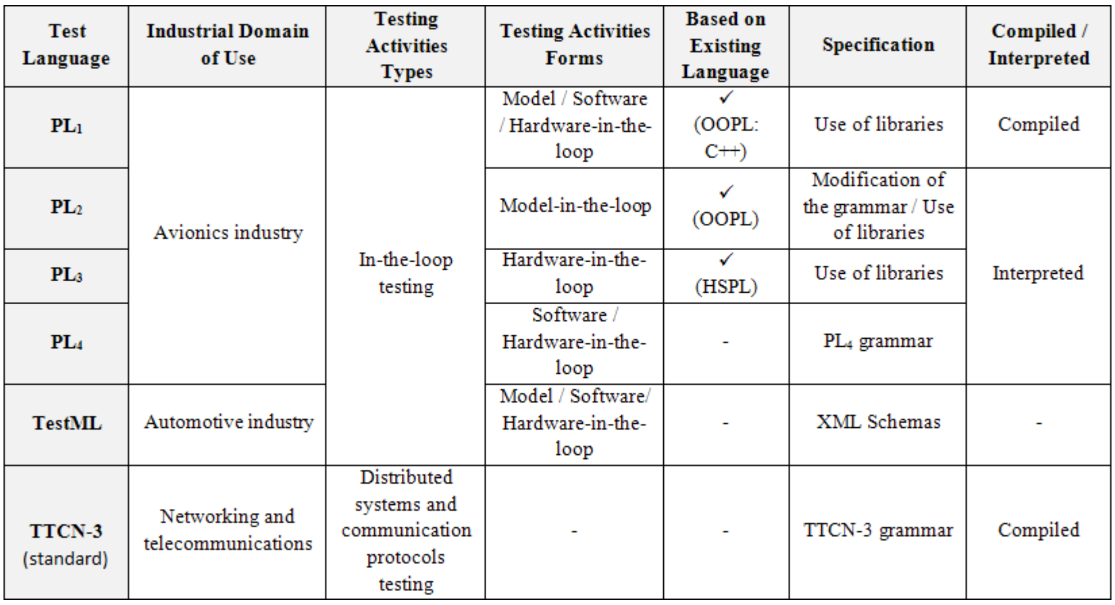
We also had access to real test specifications written in natural language (English, French).

Let us discuss the methodology that guided our test languages analysis. Our approach was to identify the features that the test languages in our sample offer, together with the similarities and the differences of the manner in which they are offered. In case a feature was not offered by some test language or offered in different manners, we contacted test engineers for clarifications. These discussions with test engineers also allowed us to identify the specific needs to which test features respond. In addition, usage problems encountered by test engineers when accessing the features were also discussed. Consequently, we were able to identify best practices and pitfalls to avoid. Moreover, our interaction with test engineers made it possible to identify new desirable features that a future test implementation solution should include.

The features we analysed are synthesised in diagrammatic views that appear at the beginning of each following section. We conclude each section with a synthesis of our main findings in the form of bullet-lists. First we discuss some generic features that the test languages exhibit (Section 4.2), continuing with the test-related features (Sections 4.3 to 4.6).

## 4.2 Generic Features

Table 4.1 gives an overview of the generic features exhibited by the test languages in our sample.



| Test Language | Industrial Domain of Use | Testing Activities Types | Testing Activities Forms | Based on Existing Language | Specification | Compiled / Interpreted |
|---|---|---|---|---|---|---|
| **PL$_1$** | Avionics industry | In-the-loop testing | Model / Software / Hardware-in-the-loop | ✓ (OOPL: C++) | Use of libraries | Compiled |
| **PL$_2$** | | | Model-in-the-loop | ✓ (OOPL) | Modification of the grammar / Use of libraries | Interpreted |
| **PL$_3$** | | | Hardware-in-the-loop | ✓ (HSPL) | Use of libraries | |
| **PL$_4$** | | | Software / Hardware-in-the-loop | - | PL$_4$ grammar | |
| **TestML** | Automotive industry | | Model / Software/ Hardware-in-the-loop | - | XML Schemas | - |
| **TTCN-3** (standard) | Networking and telecommunications | Distributed systems and communication protocols testing | - | - | TTCN-3 grammar | Compiled |

**Table 4.1:** Test Languages Generic Features

From the six test languages in Table 4.1, three are built upon existing general-purpose programming languages (GPPL), 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,
- the definition of specialized libraries.

Both options appear in Table 4.1. 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 [87]. This implementation is typically done either in C++ or Java. TestML is a mark-up language based on XML Schemas [115], and uses automata descriptions to capture the behavioural part of the test. It is not an executable language, although a MATLAB Simulink (with the extension Stateflow) [116] implementation of the automata has been proposed to demonstrate the semantics.

The reason for using GPPL-based test languages is that they offer access to a wide number reusable elements: generic functionalities (e.g., declaration of variables, logical and mathematical expressions) and tools (e.g., compilers, interpreters, integrated development environments). Moreover, test engineers familiar with existing GPPL languages can rapidly start using them. A drawback is that the test code is usually not concise, the users having to express the test concepts in terms of generic language elements.

In contrast, specific test languages offer an industrial context-specific vocabulary and are more concise than GPPL-based test languages. In order to bridge this gap, the latter category of test languages has to rely on automatic code generation or on language pre-processing techniques. In $PL_1$, code skeletons are generated, so that the test engineer may focus on writing the logic of the test. In $PL_2$, a pre-processing step allows the use of a specialised syntax. However, in practice many $PL_2$ 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 among our partners, as all the proprietary test languages in our sample propose 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 test engineers, the proposed solution should accommodate the existing custom and practice. The TTCN-3 approach could be seen as a good compromise in its domain of application, combining convenient high-level constructs and an imperative style familiar to engineers.

TTCN-3 is the only test language in our sample that has a standardized implementation, with a clear separation being offered between the generic execution kernel of the test language and all platform-specific adapters [87]. We did not have access to information concerning the implementation choices of test languages outside Cassidian Test & Services. As such, we do not discuss this issue further here.

**Key Ideas**

- ▶ Specific test languages offer a customizable, concise, high-level, domain-specific vocabulary.
- ▶ GPPL-based test languages enable the reuse of existing functionalities and tools.
- ▶ Interpreted test languages do not require a new compilation step each time the code is modified.
- ▶ Current practice, among Cassidian Test & Services partners, favours an imperative programming paradigm.
- ▶ A test platform-independent approach, with a clear separation between the test language kernel and the test platform-specific adapters is desirable.

## 4.3 Test Organization

Figure 4.2 gives an overview of the organization types that can exist for test artefacts.

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 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, although integrating it with existing intra-test organization features could be useful in order to gain in homogeneity.

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. The use of functions, allowing for code reuse by means of a concise manipulation, is only a first facility. We discuss next more complex facilities: the semantic organization of instructions (Subsection 4.3.1) and the thread of control organization (Subsection 4.3.2).

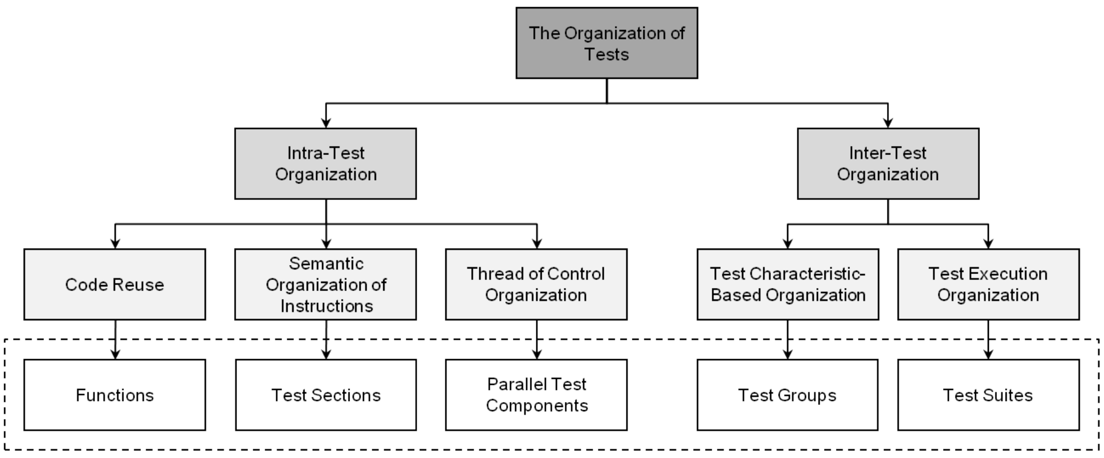

The Organization of Tests
Intra-Test Organization
Inter-Test Organization
Code Reuse
Semantic Organization of Instructions
Thread of Control Organization
Test Characteristic-Based Organization
Test Execution Organization
Functions
Test Sections
Parallel Test Components
Test Groups
Test Suites

**Figure 4.2: Tests Organization -**

### 4.3.1 Semantic Organization of Instructions

Another test organization 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. Test sections appear as distinct parts of test specifications. Test engineers use annotation-based methodologies in order to map this organization onto their test procedures: comments inside the test code separate test sections.

Useful test sections include test identification information that appears as a test header (e.g., the name of the test, the targeted SUT, the author of the test), pre-test actions (e.g., powering up the SUT, reaching a desired SUT state before the execution of the test), test behaviour (e.g., the stimulation of the SUT, the analysis of the SUT behaviour) or post-test actions (e.g., powering down the SUT).

Being a mark-up language, TestML has this form of organization but is the only language of the sample that includes it inside the language, not having to rely on comment writing methodologies. TestML has markup elements in order to stimulate the system under test, capture data and evaluate its behaviour.

As a test can be a quite complex artefact, convenient test section features are crucial for both readability and writability of the code. The concept of test sections allows test engineers to easily find the information they are looking for inside a test, by searching for the test section that is supposed/or most likely to contain it. This organization type also aids in the comprehension of a test, as all the instructions that produce a specific part of the test behaviour are regrouped. Moreover, the separation of a test into predefined test sections forces the test engineer to enter all the needed data and to not forget important information (e.g., the initialization of a test).

**Key Ideas**

- ▶ Test sections are useful for the identification of a test - acting as a test header, as well as for the organization of the behaviour of a test.
- ▶ Test sections can be defined following an annotation-based methodology or by integrating them inside the test language.

### 4.3.2 Thread of Control Organization

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 have this functionality, but there are discrepancies between what is concretely offered (Figure 4.3).

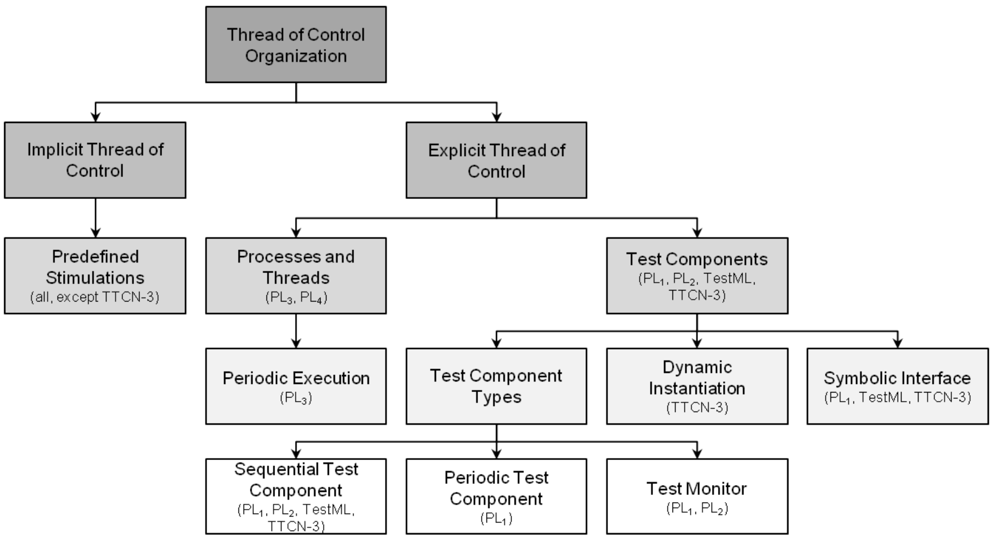

Thread of Control Organization
Implicit Thread of Control
Explicit Thread of Control
Predefined Stimulations (all, except TTCN-3)
Processes and Threads ($PL_3$, $PL_4$)
Test Components ($PL_1$, $PL_2$, TestML, TTCN-3)
Periodic Execution ($PL_3$)
Test Component Types
Dynamic Instantiation (TTCN-3)
Symbolic Interface ($PL_1$, TestML, TTCN-3)
Sequential Test Component ($PL_1$, $PL_2$, TestML, TTCN-3)
Periodic Test Component ($PL_1$)
Test Monitor ($PL_1$, $PL_2$)

**Figure 4.3: Thread of Control Organization -**

Two types of thread of control exist: explicit and implicit. Explicit thread of control offers test engineers the capacity to define their own behaviour for parallel test components and control their execution: start, stop and pause them. Implicit thread of control offers test engineers access only to predefined behavioural patterns (e.g., ramp, sine signals) by means of high-level instructions, which hide the multi-threading aspect that is delegated to the test platform run-time. For example, an instruction for applying a ramp stimulation on an application parameter launches a new background thread, which computes and sets a new value at each execution cycle. We focus here on explicit thread of control, with implicit thread of control (in the form of time-dependant SUT stimulation instructions) being discussed in Subsection 4.5.1.

$PL_3$ 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. $PL_2$ has a big main component and a set of small concurrent test monitors with a behaviour of the form: condition $\rightarrow$ action. The action can be a log or a simple input stimulation. $PL_4$ has commands to load and launch concurrent test scripts from another test script, as well as periodically execute a test script. Concurrent test scripts may be synchronized by events (e.g., `sendEvent()`, `waitEvent()` instructions). $PL_1$, 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. $PL_1$ and TTCN-3 have concrete components concurrently executing their sequential code. They both make it possible to have several instances of a generic test component, each involving its connections with other test component instances or the system under test. In addition, $PL_1$ test components can also execute their behaviour periodically, mimicking the cyclic behaviour of the system under test. This functionality is useful for the development of environment models that must have the same reactive cyclic behaviour as the SUT, as well as for the definition of complex SUT stimulations that need to be recalculated at each execution cycle (Subsection 4.5.1).

Listings 4.1 to 4.3 illustrate the concept of test component, as offered by $PL_1$ and TTCN-3.

In TTCN-3 (Listing 4.1), 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 the connections (test architecture) is dynamically determined by means of `connect()` and `map()` operations, depending on whether the connection is with other test components or the system under test. None of the proprietary test 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.

The `start()` operation (Listing 4.1) launches the execution of some behaviour by the target component. Note how the behaviour is passed as a function reference parameter.

```
/* 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         // SUT interfaces
{

/* 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 SUT */
map(self.myPort_1, system.port_1);
map(myComp_1.myPort_2, system.port_2);

/* Starting the parallel test component */
myComp_1.start(myBehavior());
...
}
```

**Listing 4.1:** Test Components in TTCN-3

In $PL_1$, generic test components are called user codes (Listing 4.2). There are also simpler (non-generic) monitors similar to the ones in $PL_2$, except that their action can also be the start of another component. $PL_1$ has no distinguished main component: from the point of view of the runtime, all user code instances and monitors are “equal citizens”, whose concurrent behaviour yields a test case. The $PL_1$ user codes communicate via shared data. 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 Listing 4.2). The language environment offers a graphical user-interface (GUI) to develop structural test elements, such as test 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 the connections is saved in an XML [73] configuration file (Listing 4.3). The topology description is thus not part of the source code of components. Neither the connect nor the map commands are present in $PL_1$, as the runtime interprets the XML file to build the configuration.

```
/* A user code has an interface and a behaviour */

/* The interface code is automatically generated from its specification
    via the GUI */
class myInterface: public userCodeInterfaceBase
{
/* Interface definition */
public:
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");
...
};
...
}
```

**Listing 4.2:** User-Codes in $PL_1$

Test 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 test engineers find it convenient to also have a predefined test monitor construct. It lets them easily express the test logic: the observation of conditions and the reactions to trigger when these conditions occur.

We have also seen that some test languages benefit from rich test development environments offering the user the possibility to define in a GUI some of the structural elements related to a test. For example, $PL_1$ users can instantiate their user codes, link their formal interfaces using such a GUI and afterwards complete the automatically generated code skeletons with the associated behaviour.

```
/* Declaration of an instance of myBehavior, which will be active from the
    beginning of the test (auto_start = "true"). */
<userCodeInstanceDeclaration userCodeInstanceName = "instanceName1"
    userCodeName = "myBehavior" auto_start = "true"/>

/* Declaration of "instanceName2" would be similar but with auto_start = "
    false" */

/* Test connections */
<userCodeLinkVariables userCodeInstanceName = "instanceName1">

/* Connection to SUT is via ICD elements */
<Link localName = "inputVariable_1" globalName = "SUT_1/ARINC_429_1/
    LABEL_1/AC_SPEED" />

/* Connection to other user-code instances is via shared auxiliary
    variables. */
<Link localName = "outputVariable_1" globalName = "auxVariable_1" />

</userCodeLinkVariables>
...
```

**Listing 4.3:** XML Topology Configuration for the $PL_1$ Runtime

### Key Ideas

- Two types of thread of control have been identified: explicit (test engineers define the parallel behaviour and control its execution) and implicit (test engineers use predefined timed instructions, such as a ramp stimulation).
- High-level constructs, like test components, hide the multi-threading implementation details.
- Inter-test-component communication may use events or shared data.
- Formal interfaces allow the reuse and the multiple instantiation of test components.
- Different types of test components are used, such as: periodic test components (for environment model development and complex SUT stimulations) and test monitors (for enhanced test logic visibility).
- The behavioural and structural concepts can be separated, with automatic code generation for the latter ones.
- The architecture of links between test components is static in our industrial context.

## 4.4 Link with System under Test Interfaces

An overview of the different features related to the link with the system under test can be found in Figure 4.4.

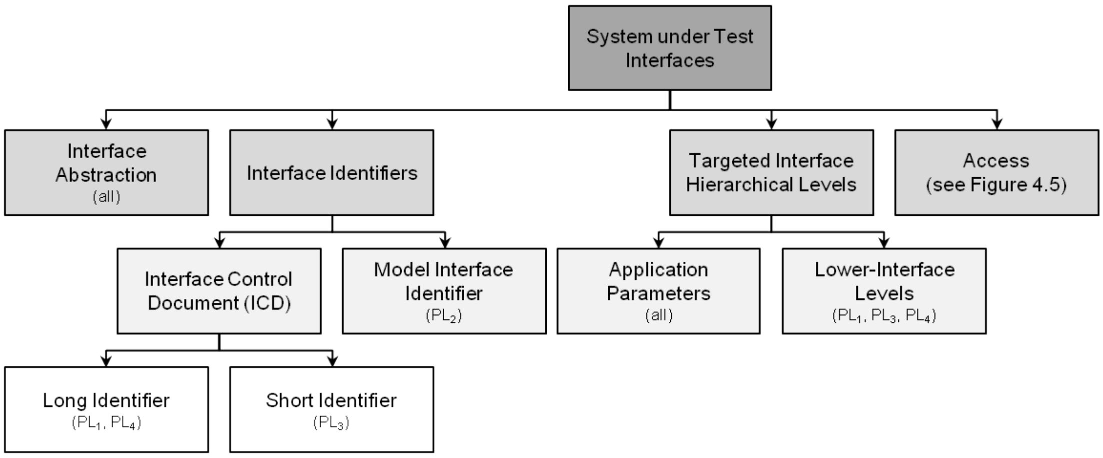

System under Test Interfaces
Interface Abstraction (all)
Interface Identifiers
Targeted Interface Hierarchical Levels
Access (see Figure 4.5)
Interface Control Document (ICD)
Model Interface Identifier (PL2)
Application Parameters (all)
Lower-Interface Levels (PL1, PL3, PL4)
Long Identifier (PL1, PL4)
Short Identifier (PL3)

**Figure 4.4: Link with the System under Test Interfaces -**

### 4.4.1 Targeted ICD Hierarchical Levels and Interface Abstraction

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 Listing 4.1. The SUT 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 [87].

The test languages for in-the-loop testing also hide the implementation details of the platform, although not in a standardized way. They access the SUT interfaces by means of ICD-derived identifiers. Their system logical interface consists of application parameters (all languages) and possibly lower-level ICD elements ($PL_1$, $PL_3$ and $PL_4$). Languages focusing on application parameters only are targeting MiL testing ($PL_2$), 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 ($PL_1$, TestML), the mapping with the logical system interface is described once and for all in an XML-based format (see Listing 4.3).

### Key Ideas

- The SUT interfaces are always abstracted via ICD-derived identifiers.
- The architecture that links test component interfaces and SUT interfaces is static in our industrial context.

### 4.4.2 SUT Interface Element Identifiers and Access

On overview of the manner in which the SUT interfaces are accessed can be found in Figure 4.5.

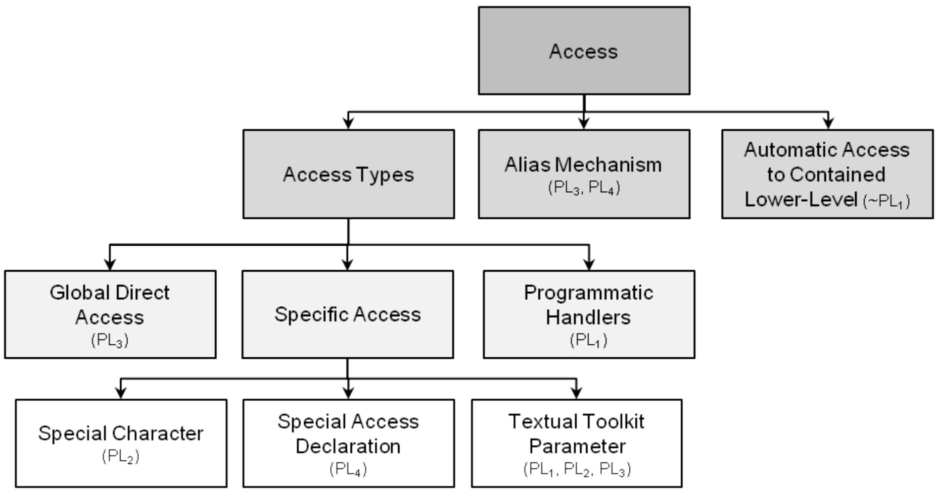

Access
Access Types
Alias Mechanism (PL₃, PL₄)
Automatic Access to Contained Lower-Level (~PL₁)
Global Direct Access (PL₃)
Specific Access
Programmatic Handlers (PL₁)
Special Character (PL₂)
Special Access Declaration (PL₄)
Textual Toolkit Parameter (PL₁, PL₂, PL₃)

**Figure 4.5: Access to the System under Test Interfaces -**

All languages for in-the-loop testing have a notion of dictionary of available application parameters. In the avionic domain, the names in the dictionary are usually built by parsing the tree-like structure of an ICD document. As mentioned previously, a name may include the complete tuple (the long identifier) or a triplet (the short identifier) that suffices to uniquely identify the application parameters. Aliasing mechanisms may be provided in order to further shorten names.

How a source code uses a name to access an application parameter varies from one language to the other. In $PL_1$, the dictionary is an external service to which data accessors are requested. This explicit request for accessors was required for performance issues. At the opposite, $PL_3$ has a global dictionary structure directly accessible in the execution environment of the test. $PL_2$ and $PL_4$ represent intermediate language design choices. The source code does not explicitly handle accessors, but a specific symbol

($PL_2$) or an access declaration instruction ($PL_4$) indicates the special access. Listings 4.4 to 4.7 show examples for the various possibilities.

Listings 4.8 and 4.9 exemplify access to other ICD hierarchical levels. We focused on $PL_1$ and $PL_4$, as $PL_3$ offers much less accessibility than the former two. In general, buses and messages are accessed for fault injection at a low level. $PL_3$ 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 analyser/exerciser).

```
/* An accessor to the linked application parameter is given at the
    creation of the user-code interface object. Other application
    parameters can also be accessed, but then the user-code behaviour
    needs to ask for an accessor. */
variableToolkit.initLink("SUT_1/ARINC_429_1/LABEL_1/AC_SPEED");

myAccessor = appParameterToolkit.getAccessor("SUT_1/ARINC_429_1/LABEL_1/
    AC_SPEED");

myAccessor.setValue(newValue);        // Writes the application parameter
myAccessor.getValue(&x, &timestamp); // Reads it in x

*/ More complex test actions use either the name: */
variableToolkit.testAction("SUT_1/ARINC_429_1/LABEL_1/AC_SPEED");

/* or the accessor as a parameter: */
otherToolkit.otherTestAction(myAccessor);
```

**Listing 4.4:** Access to SUT Application Parameters in $PL_1$

```
/* Access is granted to any application parameter, 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 application parameter identifier as a
     parameter, not the accessor */
aToolkit.testAction ("modelSUT_1/AC_SPEED");

/* Access is also given to the status variables of the system models (MiL
    testing activity) */
isStatusRunning = @modelSUT_1/Running;
```

**Listing 4.5:** Access to SUT Application Parameters in $PL_2$

```
/* 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 ]= {alias = "myAlias"}}

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

**Listing 4.6:** Access to SUT Application Parameters in $PL_3$

```
/* Access is gained by the declaration of the needed application
    parameters. An alias may be introduced by the declaration instruction.
     */
access engineerVariable 'SUT_1/ARINC_429_1/LABEL_1/AC_SPEED' as myAlias;

x = myAlias;
myAlias = 0;
aToolkit.testAction (myAlias);
```

**Listing 4.7:** Access to SUT Application Parameters in $PL_4$

```
/* A bus accessor is used to get a message accessor */
myBusAccessor = A429Toolkit.getAccessor ("SUT_1/ARINC_429_1");
myMsgAccessor = myBusAccessor.getAccessor ("LABEL_1");

/* Test actions are attached to the accessors */
MyBusAccessor.testAction ();
MyMsgAccessor.testAction ();
```

**Listing 4.8:** Access to Lower-Level ICD Hierarchical Levels $PL_1$

```
/* Bus and message have two independent accessors */
access bus 'SUT_1/ARINC_429_1' as myBusAlias;
access message 'SUT_1/ARINC_429_1/LABEL_1' as myMsgAlias;

/* Test actions are in independent toolkits */
busToolkit.testActionBus (myBusAlias);
msgToolkit.testActionMsg (myMsgAlias);
```

**Listing 4.9:** Access to Lower-Level ICD Hierarchical Levels $PL_4$

In $PL_1$ and $PL_4$, 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 $PL_1$, gaining a bus accessor does not automatically provide access to its messages, although the bus accessor is used to get a message accessor. In $PL_4$, 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. 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 $PL_1$, the configuration is described in an XML file. For each category of test resource, the $PL_1$ 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.

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 application parameters: sometimes by means of accessors, sometimes by means of a name passed as a string parameter to the action. 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 not done, which is error-prone (e.g., it is possible to perform an access demand for a non-existent SUT interface). $PL_3$ offers an interesting solution devoid of explicit accessors and string parameters: all application parameters are directly accessible as global variables in the test execution environment. This facility is however only for application parameters. When descending at lower levels of the ICD hierarchy, we once again find the string parameter solution.

**Key Ideas**

- Heterogeneous SUT interfaces access mechanisms exist, between different ICD hierarchical levels as well as for a same ICD hierarchical level.
- The structure of the ICD is only partially reflected by the test language constructs.
- Using the identifiers of ICD elements as string parameters does not allow static type-checking.
- The global access to ICD elements feature is convenient.

## 4.5 Test Language Instructions

An overview of the different features exhibited by test-related instructions can be found in Figure 4.6.

The main test-related instructions categories are: test/simulation execution control, system under test interaction (Subsection 4.5.1) and test verdict management (Subsection 4.5.2) instructions.

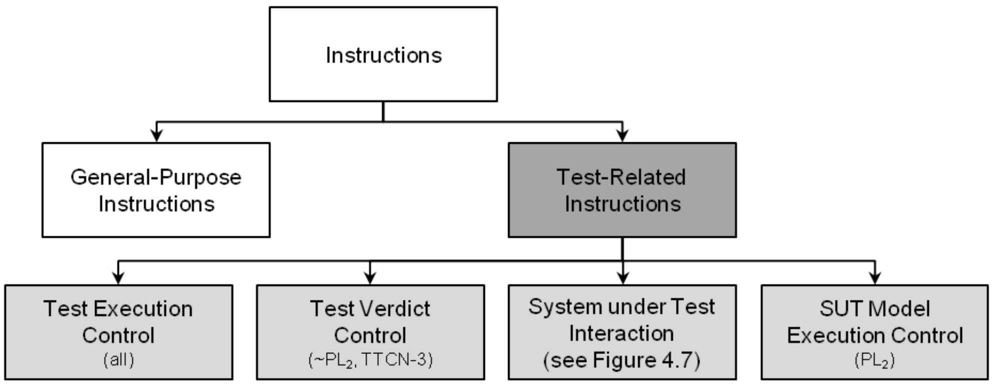

Instructions
General-Purpose Instructions
Test-Related Instructions
Test Execution Control (all)
Test Verdict Control (~PL$_2$, TTCN-3)
System under Test Interaction (see Figure 4.7)
SUT Model Execution Control (PL$_2$)

**Figure 4.6: Test Language Instructions -**

We saw an example of test execution control with the start instruction in Listings 4.1 and 4.2. All test languages have instructions to control the execution of a test. The control may also involve timing aspects such as a periodic activation of the test code. These aspects are relevant to stimulate cyclic embedded systems. Besides parallel test components, execution control may also concern the system under test, but this is only for the MiL form of testing: $PL_2$ has specific instructions to control and synchronize the execution of the system models. In the sequel, we focus on the test interactions with the SUT and on the test verdict management.

### 4.5.1 System under Test Interaction

An overview of the different features related to the SUT interaction instructions can be found in Figure 4.7.

There are interesting differences between TTCN-3 and the test languages targeting embedded systems. While the latter test languages offer a large number of different instructions, 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 application parameter interaction is to read or write a value. For this, simple assignment is offered as a syntactic facility. We saw examples

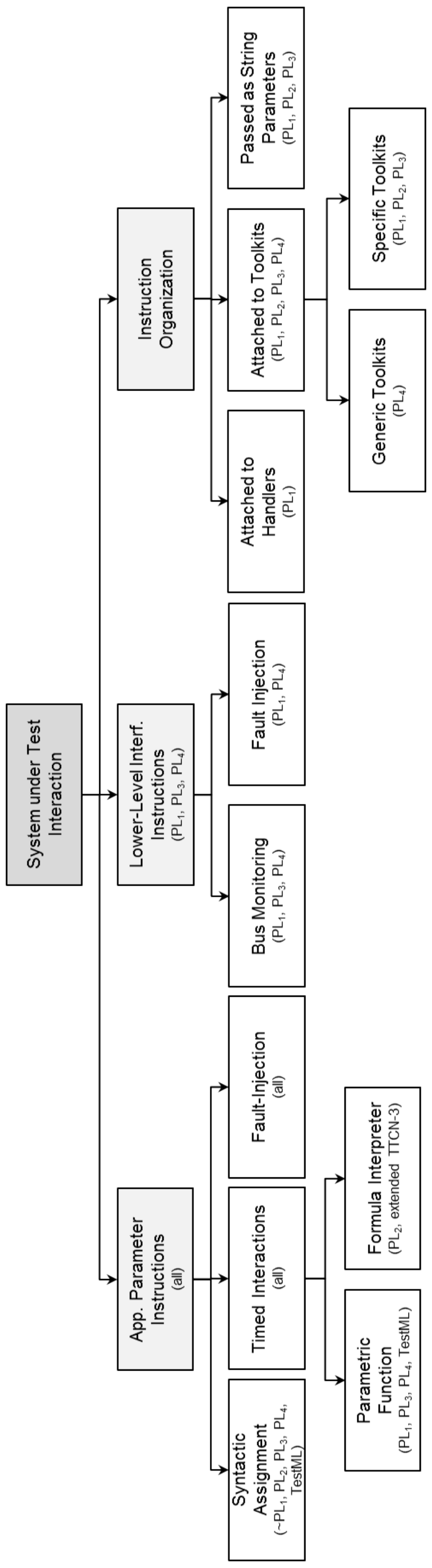

System under Test Interaction
App. Parameter Instructions (all)
Lower-Level Interf. Instructions ($PL_1$, $PL_3$, $PL_4$)
Instruction Organization
Syntactic Assignment (~$PL_1$, $PL_2$, $PL_3$, $PL_4$, TestML)
Timed Interactions (all)
Fault-Injection (all)
Parametric Function ($PL_1$, $PL_3$, $PL_4$, TestML)
Formula Interpreter ($PL_2$, extended TTCN-3)
Bus Monitoring ($PL_1$, $PL_3$, $PL_4$)
Fault Injection ($PL_1$, $PL_4$)
Attached to Handlers ($PL_1$)
Attached to Toolkits ($PL_1$, $PL_2$, $PL_3$, $PL_4$)
Passed as String Parameters ($PL_1$, $PL_2$, $PL_3$)
Generic Toolkits ($PL_4$)
Specific Toolkits ($PL_1$, $PL_2$, $PL_3$)

**Figure 4.7: System under Test Interaction Instructions -**

in Listings 4.5 to 4.7 for $PL_2$, $PL_3$ and $PL_4$. In $PL_1$, for which application parameter 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 Listing 4.2, $outputVariable_1$ 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 application parameter accessors, not the assignment. Whatever the language, a write forces a value until another value is written or the writing is explicitly disabled. 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).

In addition to the simple read and write, all in-the-loop testing languages of the sample offer a rich set of predefined application parameter interactions. They typically include stimulation patterns over time, like ramp, oscillation, triangle or square patterns. Listings 4.10 to 4.12 exemplify 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 (Listing 4.10).

```
/* A ramp signal as a parametric function */
stimulationToolkit.applyRamp(myAccessor, startValue, endValue, duration);

/* Injecting an offset on an application parameter */
variableToolkit.injection("aVariableName", "offset", offsetValue,
    listOfParameters);
```

**Listing 4.10:** Application Parameter Interactions in $PL_1$

```
/* A ramp signal with a formula interpreter */
Formula formula = new Formula("2*@t + 1");
Stimulation stimulus = new Stimulation();

stimulus.define("modelSUT_1/AC_SPEED", formula);
stimulus.start();
```

**Listing 4.11:** Application Parameter Interactions in $PL_2$

```
/* A ramp signal defined using mark-up elements */
<signal ID = "AC_SPEED" type = "double">
   <time> <unit> second </unit>
      <double><value> 4 </value></double>
   </time>
   <ramp>
      <start>
         <double><value> 1 </value></double>
      </start>
      <end>
         <double><value> 9 </value></double>
      </end>
</signal>
```

**Listing 4.12:** Application Parameter Interactions in TestML

$PL_1$, $PL_3$ 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. $PL_1$ 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). $PL_4$ sees messages as raw bit vectors and provides a generic instruction to overwrite a vector.

As we have seen, 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:

- attached to ICD elements programmatic handlers (for application parameters as well as buses or messages), where the handler was created using the string identifier:

  ```
  aHandler = getHandler("id");
  aHandler.testAction();
  ```

- grouped into specific toolkits that take the identifier or handler as a parameter:

  ```
  aToolkit.testAction("id", paramList);
  aToolkit.testAction(aHandler, paramList);
  ```

- taken as a parameter by a generic toolkit (e.g., in order to have an extensible Application Programming Interfaces (API) that could accept new test actions):

  ```
  aToolkit.do("testAction", "id");
  ```

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 [86] to add a stream port type, which would allow TTCN-3 to account for continuous data flows.

The heterogeneous view of interactions inside a language negatively impacts the readability and writability of the test code. This reflects the history of the proprietary languages, which were enriched progressively when demanded by the users. There is now a need for a coherent organization of instructions (with predefined extension points for adding new ones), which also allows for type checking at compilation 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 found in other industrial contexts, like GUI systems (classical applications in [100] and web applications in [117]): 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 element would call for its specific test actions.

**Key Ideas**

- ▶ Test languages for in-the-loop testing involve many different test interaction instructions.
- ▶ The organization of instructions within test languages is heterogeneous.
- ▶ A coherent organization policy is needed, with predefined extension points.
- ▶ Passing test actions as string parameters to generic toolkits does not allow for static type-checking (e.g., it is possible to call a test action inconsistent with the target SUT interface type).
- ▶ Attaching instructions to strongly-typed SUT interfaces is an interesting organizing principle.

### 4.5.2 Test Verdict Management

As regards test verdict management, 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. The verdict of a higher-level element (e.g., test case) is derived from the local verdicts of lower-level elements it contains (e.g., test components). Verdicts are ordered from `Error` to `Pass` with rules enforcing a conservative direction of changes: `None` > `Pass` > `Inconclusive` > `Fail` > `Error`. Consequently, a `Pass` verdict may change to `Fail` or `Inconclusive`, but an `Error` verdict never changes.

A simplified form of verdict management is also provided by $PL_2$, with global test verdict synthesis from two local possible test verdicts: `Pass` and `Fail`. A global test verdict can be `Pass`, `Fail` or `Partial` (when some local checks have failed). The other languages do not put emphasis on verdicts, because test evaluation is usually not performed on-line by an automated test oracle. SiL and HiL test platforms include detailed data recording facilities, and the recorded data is analysed off-line by test engineers with the aid of visualization tools.

**Key Ideas** Verdict management and automatic test verdict synthesis is quasi-completely lacking from the proprietary test languages.

## 4.6 Time Management

An overview of the different features related to time management can be found in Figure 4.8.

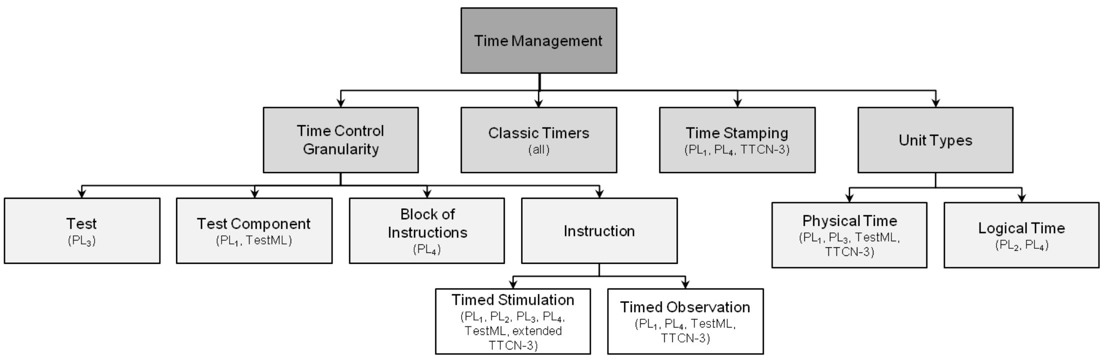

Time Management
Time Control Granularity
Classic Timers (all)
Time Stamping (PL1, PL4, TTCN-3)
Unit Types
Test (PL3)
Test Component (PL1, TestML)
Block of Instructions (PL4)
Instruction
Physical Time (PL1, PL3, TestML, TTCN-3)
Logical Time (PL2, PL4)
Timed Stimulation (PL1, PL2, PL3, PL4, TestML, extended TTCN-3)
Timed Observation (PL1, PL4, TestML, TTCN-3)

**Figure 4.8: TimeManagement -**

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 [84, 86], with a time-stamping of communication events and a timed control of events, although they are not yet part of the standard.

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 (Listings 4.10 to 4.12). Data observation can also be made time-dependent, as in the $PL_1$ and $PL_4$ instructions found in Listing 4.13:

```
/* Waiting for an event in PL_1 */
eventToolkit.waitValue(myAccessor, expectedValue, tolerance, timeout);

/* Waiting for an event in PL_4 */
eventToolkit.waitCondition(myCondition, timeout, checkPeriod);
```

**Listing 4.13:** Waiting for an Event in $PL_1$ and $PL_4$

Note that $PL_1$ and $PL_4$ run on top of real-time operating systems.

Usually, time is expressed in physical units (e.g., in seconds). In $PL_2$ and $PL_4$, we found some instructions with logical units (i.e., a number of cycles). This is quite natural for the MiL testing usage of $PL_2$, 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 a 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.

Listings 4.14 to 4.16 show the facilities offered by $PL_1$, $PL_3$ and $PL_4$. Depending on the language, timed test execution control is applied at a different level of granularity: from blocks of instructions to entire tests.

The finest-grained control is in $PL_4$. It is at the level of blocks of instructions, where a test can contain blocks to be executed in bounded time on tick reception from the synchronization service (Listing 4.16). Currently this functionality provided by $PL_4$ is deprecated, but was presented in the version of the user manual we analysed. A problem is that the $PL_4$ instructions do not guaranty that there is no tick between the end of a synchronous block (i.e., `tick.complete()`) and the beginning of the next (i.e., `tick.wait()`). Discussions with test engineers concluded that an enhanced time-triggered sequential execution could be helpful. It would allow a precise step-by-step control over the interaction with SUT interfaces. Test behaviour could be "quasi-synchronized" with the cyclic behaviour of the SUT. In order to address these issues, we propose a possible solution in the form of a new type of test component (see Subsection 5.4.3).

In $PL_1$, 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.

$PL_3$ does not have the notion of component, but a test can be repeated periodically. When test execution is not deterministic, this allows for the execution of various possible interleavings.

We did not include TestML examples in Listings 4.14 to 4.16. Compared to the others, this language would have a very different flavour 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 behaviour. This is where facilities like periodic activation and bounded time execution prove useful.

### Key Ideas

- ▶ Time can be measured in physical and logical units, for respectively hardware-in-the-loop and software/model-in-the-loop testing.
- ▶ Predefined timed stimulations and wait instructions are offered.
- ▶ Time management is achieved at different granularity levels, ranging from blocks of instructions to entire tests.
- ▶ Periodic activation and bounded time execution facilities are offered.
- ▶ A step-by-step control of the interaction with the SUT would be useful, for more precise testing of real avionics equipment.

With this section we have finished the analysis of the test language features. We continue with a synthesis of the results of our analysis in a table of guiding principles for our meta-modelling work.

```
/* 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"/>
```

**Listing 4.14:** Timed Control of Test Execution in $PL_1$

```
/* A test can be periodic */

testIdentification = {... testPeriod = 100.0;}
```

**Listing 4.15:** Timed Control of Test Execution in $PL_3$

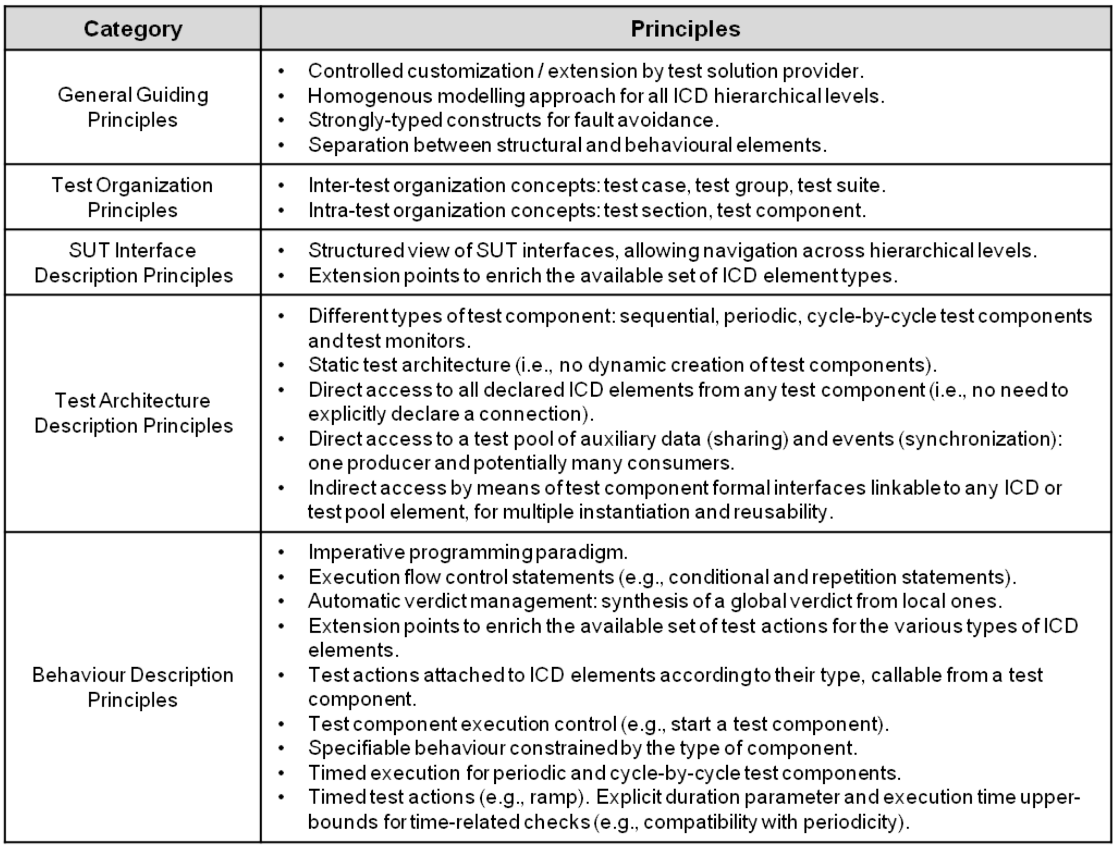
```
/* Execution control is for blocks of test instructions. It uses a tick
    service. */

/* Frequency is 50 hertz */
tick.register(50);

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

/* 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();
}

/* From now on, no tick synchronization. */
tick.stop();
```

**Listing 4.16:** Timed Control of Test Execution in $PL_4$

## 4.7 Meta-Modelling Guiding Principles

Table 4.2 presents the list of principles that guided our formalization of the different domain-specific concepts (e.g., test case, test component, test verdict) inside the test meta-model. This list is derived from the key ideas found at the end of each previous subsection.

We briefly discuss here the “General Guiding Principles”. They are declined into more specific principles found in the rest of the table. For the reduction of the level of heterogeneity we consider that a controlled customization/extension of the test development solution is desirable. Consequently, new types of SUT interface types and associated test actions can be added at predefined extension points, when needed. We propose a homogeneous access approach for the interfaces of the SUT, based on the hierarchical organization of its ICD: the structure of the ICD is projected onto object structures. Our proposal also deals with fault avoidance, by using only strongly typed constructs, replacing current practice where SUT interface identifiers and test actions are sometimes passed as string parameters. Finally, the structural (e.g., test case, test component) and behavioural (e.g., calling of test actions on SUT interfaces) elements are separated, in order to allow the development of a graphical editor for the structural ones and a textual editor for the behavioural ones.



| Category | Principles |
|---|---|
| General Guiding Principles | • Controlled customization / extension by test solution provider.<br>• Homogenous modelling approach for all ICD hierarchical levels.<br>• Strongly-typed constructs for fault avoidance.<br>• Separation between structural and behavioural elements. |
| Test Organization Principles | • Inter-test organization concepts: test case, test group, test suite.<br>• Intra-test organization concepts: test section, test component. |
| SUT Interface Description Principles | • Structured view of SUT interfaces, allowing navigation across hierarchical levels.<br>• Extension points to enrich the available set of ICD element types. |
| Test Architecture Description Principles | • Different types of test component: sequential, periodic, cycle-by-cycle test components and test monitors.<br>• Static test architecture (i.e., no dynamic creation of test components).<br>• Direct access to all declared ICD elements from any test component (i.e., no need to explicitly declare a connection).<br>• Direct access to a test pool of auxiliary data (sharing) and events (synchronization): one producer and potentially many consumers.<br>• Indirect access by means of test component formal interfaces linkable to any ICD or test pool element, for multiple instantiation and reusability. |
| Behaviour Description Principles | • Imperative programming paradigm.<br>• Execution flow control statements (e.g., conditional and repetition statements).<br>• Automatic verdict management: synthesis of a global verdict from local ones.<br>• Extension points to enrich the available set of test actions for the various types of ICD elements.<br>• Test actions attached to ICD elements according to their type, callable from a test component.<br>• Test component execution control (e.g., start a test component).<br>• Specifiable behaviour constrained by the type of component.<br>• Timed execution for periodic and cycle-by-cycle test components.<br>• Timed test actions (e.g., ramp). Explicit duration parameter and execution time upper-bounds for time-related checks (e.g., compatibility with periodicity). |

**Table 4.2:** Test Meta-Modelling Guiding Principles

Table 4.2 includes all interesting features we found in our analysis of test languages. In addition to them, we propose two new ones that have been deemed useful.

The first new feature is the cycle-by-cycle test component ("Test Architecture Description Principles", bullet 1) that is usable for specifying precise test behaviour with respect to the cyclic behaviour of the SUT. The component has blocks of instructions to be executed at consecutive cycles.

The second new feature is based on an issue we identified with the existing proprietary test languages: they lack control over the type of behaviour that test engineers can specify inside the different types of test artefacts (e.g., test case, test component). No control exists over the execution flow (e.g., repetition, conditional statements) and SUT interaction test actions that can be called. For example, unbounded repetition statements and test actions exhibiting durations (e.g., ramp, sinus) higher than the period of a periodic test component can be called inside the periodic test component. Moreover, several test components are allowed to call test actions with side-effects (i.e., reading the value of an application parameter is side-effect-free, while writing it has side effects) on ICD elements. This can lead to clashes and unexpected behaviour. Such test specification problems should be detectable before execution. Consequently, our second new feature is to constrain and validate the test behaviour before its execution, in order to prevent inconsistencies ("Behaviour Description Principles", bullet 7).

Chapter 5 presents the test meta-model we defined. It shows the manner in which the aforementioned guiding principles where taken into account by our meta-modelling work.

## 4.8 Conclusion

In this chapter we have analysed six test languages. We focused on four proprietary languages (from $PL_1$ to $PL_4$) that are currently employed in the avionics industry, for the in-the-loop testing of avionics 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. To the best of our knowledge, no such analysis has been performed before. A threat to the validity of our analysis is the fact that our sample of test languages is limited. In any case, it is representative of the partners of Cassidian Test & Services. Moreover, we had the opportunity to discuss test language issues while taking part in an industrial project assembling various stakeholders in the field of avionics.

Our analysis focused on a number of features and the way they are offered by each test language. It confirmed the heterogeneity of test languages: not all test languages offer the same features, shared features are offered in different manners, and sometimes even a same language offers one feature 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.

We consider that our test languages analysis can be useful for the stakeholders in the avionics domain. If they already have a test language, then they can improve it by taking into account the best practices we identified. If they wish to define a new test language, our analysis has revealed the set of domain-specific concepts that need to be considered.

The analysis of test languages convinced us that existing standardized test languages used in other fields are not easily portable to our domain. Test languages used in hardware testing (ATLAS [112], ATML [113]) target mostly structural electronic circuitry defects that are detected by applying electrical signals at different places in the circuit. In contrast, the in-the-loop testing of avionic embedded systems targets the functional logic of the system, implemented by software components. TTCN-3 [82] targets mostly open-loop distributed systems, where the asynchronous sending of a few messages triggers the functional activity of an otherwise quiescent system. This allows TTCN-3 to abstract all interactions with the system under test into a small number of core instructions. This approach does not correspond to our industrial context, where the system under test exhibits a cyclic behaviour and where the number of instructions is high and dependent on the type of communication mean (e.g., the AFDX and ARINC 429 each have their own specific possible interactions).

Our analysis convinced us that the multiplicity of in-house solutions should be addressed at a higher-level, the one of test concepts. This was the foundation for proposing 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 [92], such as meta-modelling techniques and model-to-code transformations.

In order to pursue our approach, we developed a meta-model of in-the-loop tests that captures the language concepts we identified as of interest. Our analysis of the sample of test languages has lead to the identification of a number of test meta-modelling guiding principles. Table 4.2 presents a synthesis of these guiding principles, that were used when defining the test meta-model. We present the test meta-model next, in Chapter 5.

# 5

# Test Meta-Model

This chapter presents the test meta-model we defined. It is specific to the in-the-loop testing of avionics systems, integrating a rich set of domain-specific concepts extracted from our analysis of test languages (Chapter 4). The definition of the test meta-model followed the guiding principles shown in Figure 4.2. We presented the test meta-model in an article accepted at an international conference [118].

The test meta-model allows for customization and maintenance of the testing solution, by providing a clear separation between the test solution provider and user sections, with predefined extension points. Customization refers to the choice that the test solution provider has over the functionalities that a test solution user has access to. Maintenance refers to the fact that the test solution provider can add/modify/remove functionalities in a controlled way. The test meta-model also keeps a separation between structural and behavioural concepts. Structural concepts are entered using a graphical editor, while a textual editor is offered for the behavioural concepts. Still, all elements are consistently integrated, with type-dependent restrictions for the behaviour attached to the structure. Overall, the model-driven approach should contribute not only to homogenization at an abstract level, but also to fault avoidance. Some programming errors are avoided by construction, or detected by automated checks performed on the test models.

In Section 5.1 we briefly introduce the meta-modelling language we used: Eclipse Modeling Framework (EMF) Ecore. The following sections present the constituents of the meta-model, an overview of which is given in Figure 5.1. We first present the separation between the test solution provider and the test solution user sections in Section 5.2. Afterwards, we discuss the test solution user section and the test context in Section 5.3. Its high/low-level structural concepts (e.g., test case, test component, test section for a test case) are presented in Sections 5.4 and 5.5 respectively, while its behavioural concepts (e.g., test action call statements) are discussed in Section 5.6.

Finally, we show the mixed (graphical and textual) test editors that can be defined in Section 5.7. Section 5.8 presents the conclusion.

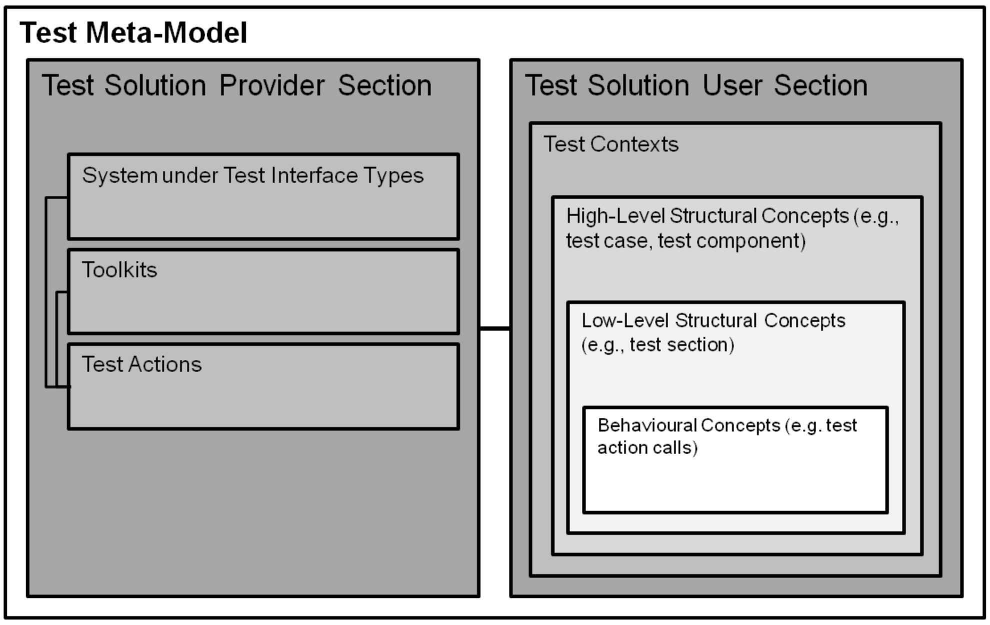

Test Meta-Model
Test Solution Provider Section
System under Test Interface Types
Toolkits
Test Actions
Test Solution User Section
Test Contexts
High-Level Structural Concepts (e.g., test case, test component)
Low-Level Structural Concepts (e.g., test section)
Behavioural Concepts (e.g. test action calls)

**Figure 5.1: Test Meta-Model - High-Level Overview -**

## 5.1 Eclipse Modeling Framework (EMF) Ecore

We retained EMF Ecore [119] as the meta-modelling language. The EMF project is a modelling framework with facilities for building tools and other applications based on a structured data model. EMF has a distinction between the meta-model and the actual model. The meta-model describes the structure of the model. A model is then the instance of this meta-model. The manner in which a meta-model constraints the definition of models is similar to the manner in which a language grammar constrains the writing of code in the given language. From a model specification serialized in XML Metadata Interchange (XMI) [120], EMF provides tools and runtime support to produce a set of Java classes for the model, along with a set of adapter classes that enable viewing and command-based editing of the model, and a basic editor. Moreover, EMF allows access to associated tools to produce specialized graphical editors (Graphical Modeling Framework (GMF) [121], Graphiti [122]), textual editors (Xtext [123]), checkers (Object Constraint Language (OCL) [124]), as well as code generators (Acceleo [125], Xpand [126]).

An EMF Ecore meta-model comprises the following types of elements, similarly to a UML [93] Class Diagram:

- `EPackage`: represents a set of classes,
- `EClass`: represents a class, with zero or more attributes, operations or references,
  - `EAttribute`: represents an attribute, which has a name and a type,
  - `EOperation`: represents an operation, which has a name, a type and parameters,
  - `EReference`: represents one end of an association between two classes, it has flag to indicate if it represent a containment and a reference class to which it points,
- `Inheritance`: refers to the ability of one class (child class) to inherit the identical functionality of another class (super class), and then add new functionality of its own,
- `EEnum`: represents a set of literals,
  - `EEnumLiteral`: represents a literal,

Our test meta-model integrates a rich set of concepts formalized in 190 EClass elements. Their characteristics and relations are formalized using 340 EAttribute and EReference elements, as well as 18 EEnum elements. For some of the concepts, the traditional two-level (meta-model and model level) instantiation step offered by EMF Ecore was not sufficient. Let us take the example of the component concept. We actually need three levels: the abstract test component concept, several user-defined components (each with its behaviour), and for each component various instances in the test architecture. In such cases, we used the Type Object design pattern [127]. It consists in decoupling instances from their type. Hence, we have two EClass elements at the meta-model level (e.g., component type and component instance) with an association between the two. This method increases the size of the meta-model but gives the required flexibility to populate the model level. The same pattern is used for our customization facilities, where SUT interface types need to be created by the test solution provider and afterwards be instantiated several times by test engineers for their specific SUT.

## 5.2 ProviderData and UserData

The root of the test meta-model is the `Database` (Figure 5.2). It contains the `ProviderData` and `UserData`, which separate the elements that are defined by the test solution provider from those defined by the test solution user (test engineers). This is a first important meta-modelling choice: the test solution user receives a pre-instantiated model from the test solution provider, where only the ProviderData section is filled in. Its elements are accessible to the test solution user inside the UserData section. The test solution provider can create different variants for its customers, as well as update existing ones. The test solution provider section offers extension points for SUT types, SUT interface types, toolkits and test actions that they possess, as existing elements of these types are abundant and new ones can appear.

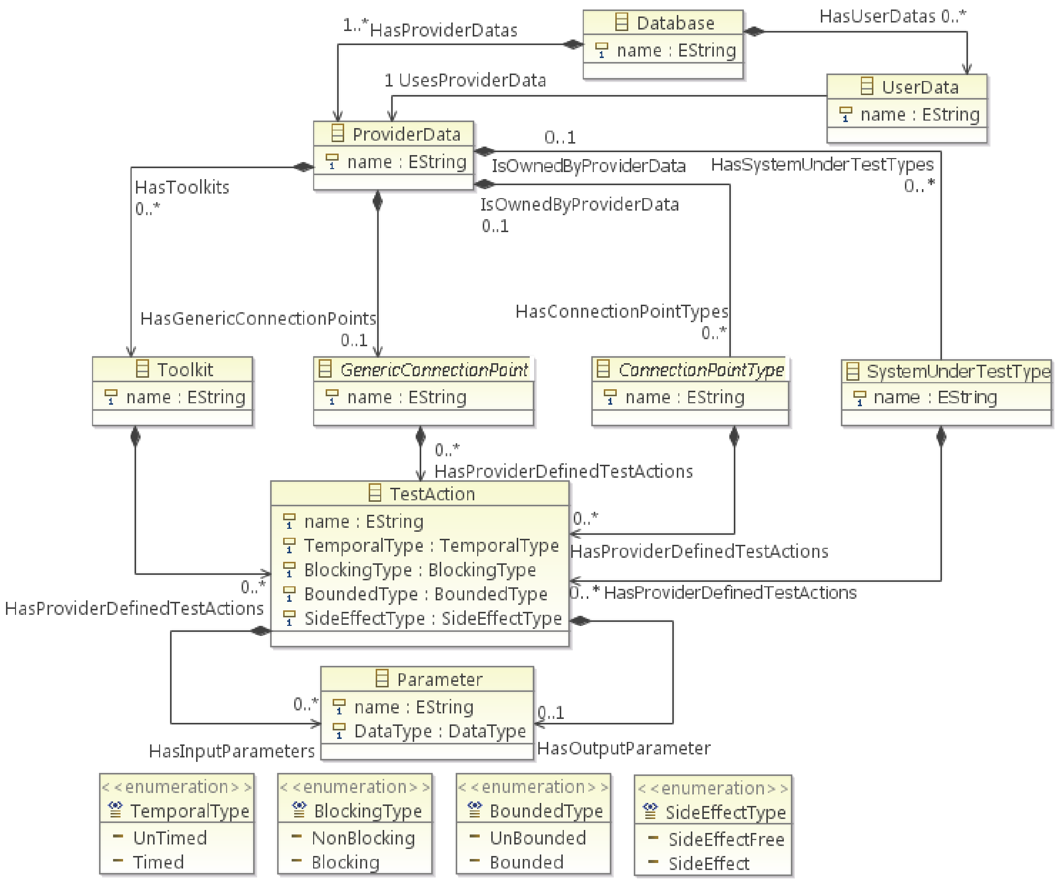


**Figure 5.2: Database Excerpt -**

Constraints are imposed on the manner in which new functionalities are added, contributing to fault avoidance: the test engineer is obliged to respect the methodology proposed by the test meta-model, being less prone to render it incoherent. A System-UnderTestType is used to assemble test actions that are specific to SUT types (e.g., test actions specific to the virtual avionics SUT type). A GenericConnectionPoint is used to assemble test actions that are common to all the elements at a particular ICD level (e.g., test actions common to all physical bus types). A ConnectionPointType is used to assemble specific test actions (e.g., test actions specific to the AFDX physical bus type). Their different specializations (e.g., `GenericPhysicalBus`, `LogicalBusType`) are shown in Figure 5.3.

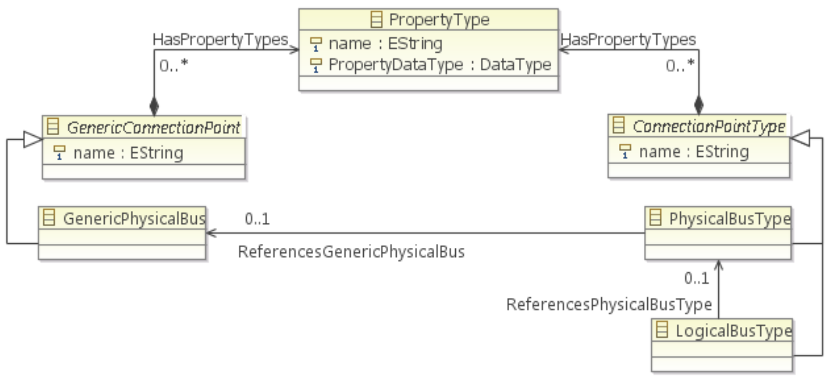


**Figure 5.3: GenericPhysicalBus/PhysicalBusType Excerpt -**

Any `GenericConnectionPoint` and `ConnectionPointType` element can possess `PropertyType` elements (Figure 5.3). Let us take the example of an AFDX Virtual Link. Virtual Links are unidirectional logic path from a source end-system to all of the destination end-systems. Unlike a traditional Ethernet switch which switches frames based on the Ethernet destination or MAC address, AFDX routes packets using a Virtual Link identifier. Inside our test model we can define an AFDX Virtual Link `LogicalBusType` that owns the Source_IP and Destination_IP `PropertyType` elements (Figure 5.4). It will be shown later on, in Subsection 5.4.1, the manner in which these types are instantiated, in conformance to the Type Object design pattern philosophy.

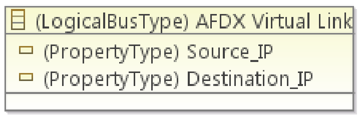

(LogicalBusType) AFDX Virtual Link
(PropertyType) Source_IP
(PropertyType) Destination_IP

**Figure 5.4: Test Model Level - ProviderData Example -**

A number of EReference elements were introduced inside the test meta-model in order to define the links between the different ICD hierarchical levels (Figure 5.3). This part of the test meta-model can be seen as an ICD meta-model. For example, an instance of a complex physical bus type (e.g., AFDX) can possess several instances of logical buses (e.g., AFDX Virtual Links). This relation is defined through the `ReferencesPhysicalBusType` EReference between an AFDX `PhysicalBusType` and the corresponding AFDX Virtual Link `LogicalBusType`. When test engineers instantiate these elements in order to define the ICD of a specific SUT, verifications are performed with the aid of OCL in order to verify that the instances respect the constraints defined on the types. For example, we can verify that an AFDX Virtual Link `LogicalBusInstance` is owned by an AFDX `PhysicalBusInstance`.

The `ReferencesGenericPhysicalBus` relation is employed to allow a specific bus type to know the generic bus test actions. All test actions referring to interactions with the SUT (e.g., sending a message, setting the value for an application parameter) are distributed inside these `GenericConnectionPoint` and `ConnectionPointType` structures. For example, as all physical buses can implement the stopEmission test action, then this test action should be attached to a `GenericPhysicalBus`. Afterwards, any specific physical bus, such as the AFDX or ARINC 429 `PhysicalBusType` elements, would reference the generic physical bus. Additional test actions, coming from external utilities, are distributed inside `Toolkit` structures (e.g., fileToolkit with write ()).

Figures 5.2 and 5.3 show only an excerpt of the test meta-model with regard to the specializations of the `GenericConnectionPoint` and `ConnectionPointType` elements and the relations that exist between them. The test meta-model actually covers all the hierarchical levels present in an ICD:

- `GenericConnector/ConnectorType`,
- `GenericPin/PinType`,
- `GenericPhysicalBus/PhysicalBusType`,
- `GenericLogicalBus/LogicalBusType`,
- `GenericMessage/MessageType`,
- `GenericMessageField/MessageFieldType`,
- `GenericApplicationParameter/ApplicationParameterType`.

Notice in Figure 5.2 the different EAttribute elements possible for a `TestAction`. They are useful for timing calculations and clash detection.

The `TemporalType` refers to whether a `TestAction` has a duration or not (e.g., a `Timed` ramp() stimulation versus a `UnTimed` setValue()). When test engineers call a `Timed`::`TestAction` inside a TestComponent, the duration parameter is rendered explicit.

The `BoundedType` refers to whether a `TestAction` can be called with an execution time upper-bound.

The `BlockingType` refers to whether the execution of a `TestAction` pauses the execution of the caller `TestComponent`. A `Blocking`::`TestAction` interrupts the execution of the caller `TestComponent`, with the `TestComponent` continuing its execution only after the execution of the `TestAction` is finished. A `NonBlocking::TestAction` executes in parallel with the caller `TestComponent`.

Finally, the `SideEffectType` refers to whether the execution of a `TestAction` produces side-effects on the element on which it is called. For example, a `SideEffect::TestAction` is the writing of a new value of an application parameter, while a `SideEffectFree::TestAction` is the reading of the value of an application parameter.

## 5.3 TestContext

The `UserData` contains `TestContext` instances. The concept of `TestContext` is inspired from the one proposed in the UML Testing Profile (UTP, [94]). It serves as a container for a collection of test cases applied to a SUT, together with an architecture of test components.

Figure 5.5 shows a high-level view of the `TestContext`.

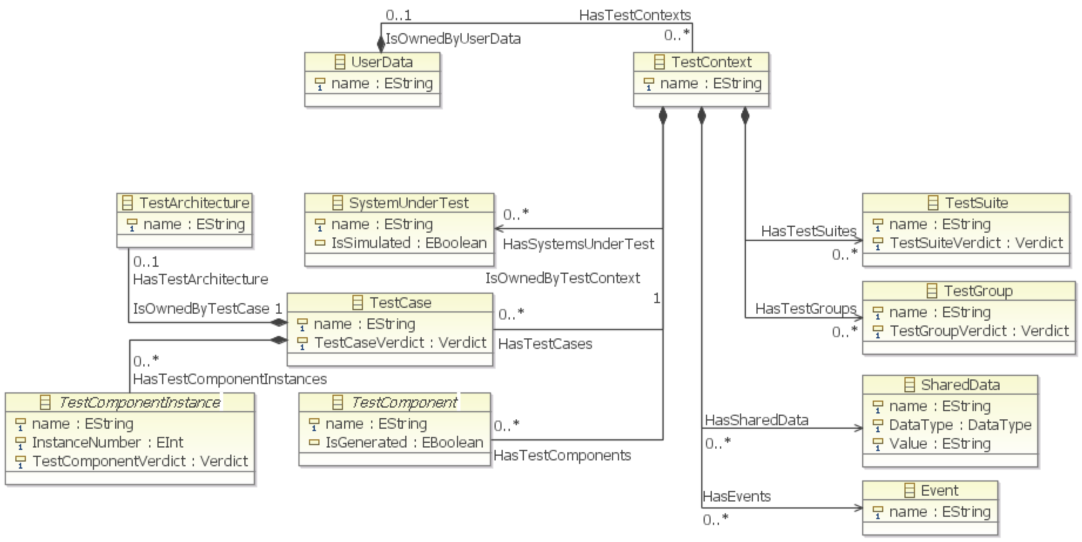


**Figure 5.5: TestContext Excerpt -**

The `SystemUnderTest` describes the interfaces of the tested entity according to its ICD.

The `TestCase` controls the parallel execution of `TestComponent` elements. A test component can be instantiated several times inside a test case, via `TestComponent-Instance` elements. The instantiation relation is not present in Figure 5.5, as it is defined on the specializations of the shown concepts (e.g., a `SequentialTestComponent-Instance` instantiates a `SequentialTestComponent`). Figure 5.5 does not present all of the links between the test case and other elements contained by the test context (e.g., the EReference `IsSequencedByTestSuites` linking `TestCase` and `TestSuite` elements is not shown).

Test components interact with the SUT. They can also interact with each-other, by means of `SharedData` and `Event` elements globally declared in the test context.

For the test architecture, we defined a policy of one producer and potentially many consumers. The `TestArchitecture` associated with a test case determines which test component instance produces a global shared data or event. It also links the formal parameters (if any) of a test component to the SUT, shared data or events.

Test cases can be regrouped within either a `TestSuite` (for execution order definition) or a `TestGroup` (for grouping tests that share a common property).

Conceptually, the test context is actually divided into three hierarchical levels:

- high-level structural concepts (e.g., `TestCase`, `TestComponent`),
- low-level structural concepts (e.g., `TestSection` in a `TestCase`),
- behavioural concepts (i.e, `TestCaseStatement`, `TestComponentStatement`).

These boundaries are useful for the definition of the mixed (graphical and textual) test model development environment: structural elements are better described inside a graphical editor, while behavioural ones with a textual editor. Both types of editors are customizable: several graphical and textual representations can be defined on top of the same test meta-model. For example, in the textual editor, the concrete syntax of behavioural elements can be customized to accommodate the user's habits.

## 5.4 High-Level Structural Concepts

We discuss in this section the different high-level structural concepts in more detail:

- `SystemUnderTest` (Subsection 5.4.1),
- `TestCase` (Subsection 5.4.2),
- `TestComponent` (Subsection 5.4.3) and the verdict management (Subsection 5.4.4),
- `TestArchitecture` (Subsection 5.4.5),
- `TestGroup` and `TestSuite` (Subsection 5.4.6).

### 5.4.1 SystemUnderTest

The `SystemUnderTest` is used to describe the interfaces of the tested entity, according to its ICD, by comprising `ConnectionPointInstance` elements (Figure 5.6). Notice how the ICD structure of hierarchical levels is reproduced inside the tree-like structure of the `SystemUnderTest`. While the `ProviderData` corresponded to an ICD meta-model, the `SystemUnderTest` corresponds to an ICD model.

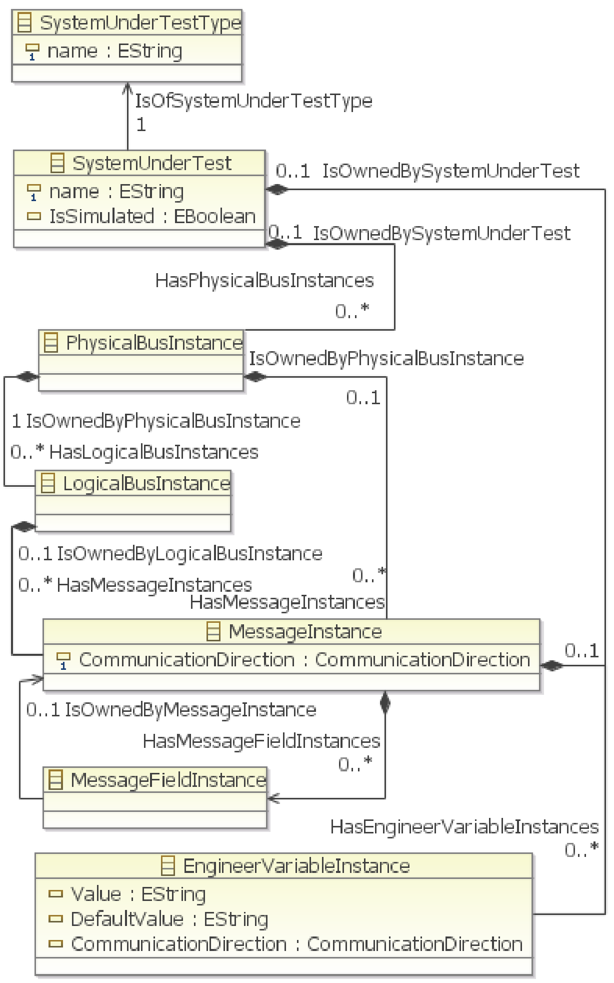


**Figure 5.6: System Under Test Excerpt -**

`ConnectionPointInstance` elements are of the `ConnectionPointType` variants defined previously in the `ProviderData` (Figure 5.7). For example, we can define an AFDX_VL_1 `LogicalBusInstance` that is of the AFDX Virtual Link `LogicalBusType`. This applies also to `PropertyInstance` elements that are of the `PropertyType` variants. For example, the AFDX_VL_1 `LogicalBusInstance` can possess the Source_IP and Destination_IP `PropertyInstance` elements corresponding to two `PropertyType` elements with the same name (Figure 5.8). This approach is coherent with the Type Object design pattern.

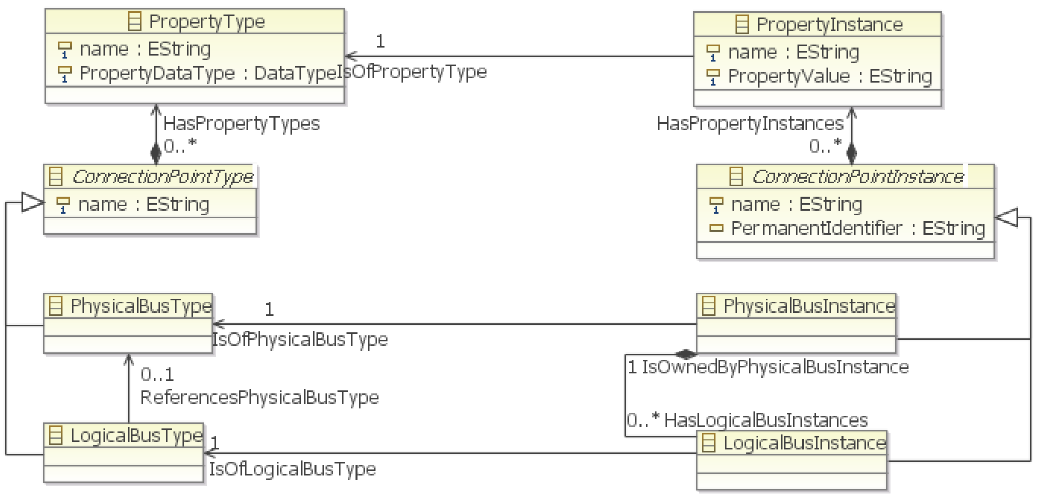


**Figure 5.7: Connection Point Instance Excerpt -**

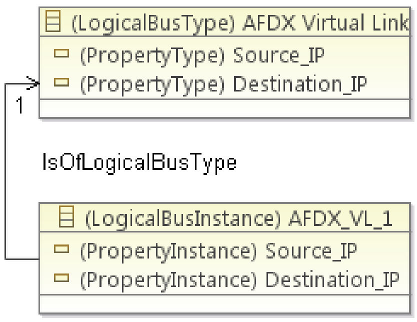

(LogicalBusType) AFDX Virtual Link
(PropertyType) Source_IP
(PropertyType) Destination_IP
1
IsOfLogicalBusType
(LogicalBusInstance) AFDX_VL_1
(PropertyInstance) Source_IP
(PropertyInstance) Destination_IP

**Figure 5.8: Test Model Level - AFDX Virtual Link Example -**

Not all connection point instance specializations are shown in Figure 5.7, but the test meta-model covers all ICD hierarchical levels:

- `ConnectorInstance`,
- `PinInstance`,
- `PhysicalBusInstance`,
- `LogicalBusInstance`,
- `MessageInstance`,

- `MessageFieldInstance`,
- `ApplicationParameterInstance`.

### 5.4.2 TestCase

A `TestCase` is an executable element whose execution status is managed either automatically by the test model execution environment via test suites, or manually by a test engineer. A test case controls the execution of test component instances (which are executable elements as well). It can also own a test architecture that defines the mappings between the interfaces of the test components and those of the system under test, the events and the shared data (Figure 5.5) A test case cannot interact with the SUT. It can possesses an `InitializationSection` in which initial values are set for the different `SharedData` elements it employs.

### 5.4.3 TestComponent

A `TestComponent` is an element that can be instantiated multiple times inside a test case, as well as reused across several test cases, yielding several `TestComponent-Instance` elements. The test component itself is not a directly executable element, but its instances are and each one of them possesses its own execution thread. Only test components are capable of interacting with the SUT. A Test component can directly access `SystemUnderTest`, `SharedData` and `Event` elements, as well as indirectly through means provided by its interfaces that we discuss later on.

The `isGenerated` EAttribute is used in order to identify test components implemented outside the test model, but whose execution needs to be controlled by a test case. For example, one can control the execution of an externally implemented complex behaviour (e.g., C++/Java class) inside the test model, by assigning it to a test component. Thus the external behaviour can be instantiated multiple times as well. This functionality is useful for example in order to control environment models coded outside of the test model.

Four types of `TestComponent` element types have been defined (Figure 5.9), depending on the behaviour to be specified:

- `TestMonitor`,
- `SequentialTestComponent`,
- `PeriodicTestComponent`,
- `CycleByCycleTestComponent`.

A `TestMonitor` has a simple behaviour of the form $logicalCondition \rightarrow behaviour$. Notice that the life duration of a test monitor is explicit (`ActivatedDuration`), as well as the frequency with which the $logicalCondition$ is evaluated (`PredicateCalculation-RefreshDuration`).

A `SequentialTestComponent` has a behaviour executed only once.

A `PeriodicTestComponent` has a behaviour that is executed periodically.

A `CycleByCycleTestComponent` has several behaviours, each one being executed during one or several consecutive cycles of the SUT. The periodic and cycle-by-cycle test components are specific to our industrial context, as they can be "quasi-synchronized" with the execution cycles of the SUT.

Notice in Figure 5.9 the `Period/CycleDuration` and `TimeMeasurementUnit` EAttribute elements. As these elements belong to the test component instances, it is possible to instantiate a periodic test component several times, with different values for its period duration. Also notice in the same figure the decoupling of component types and instances, based on the Type Object design pattern.

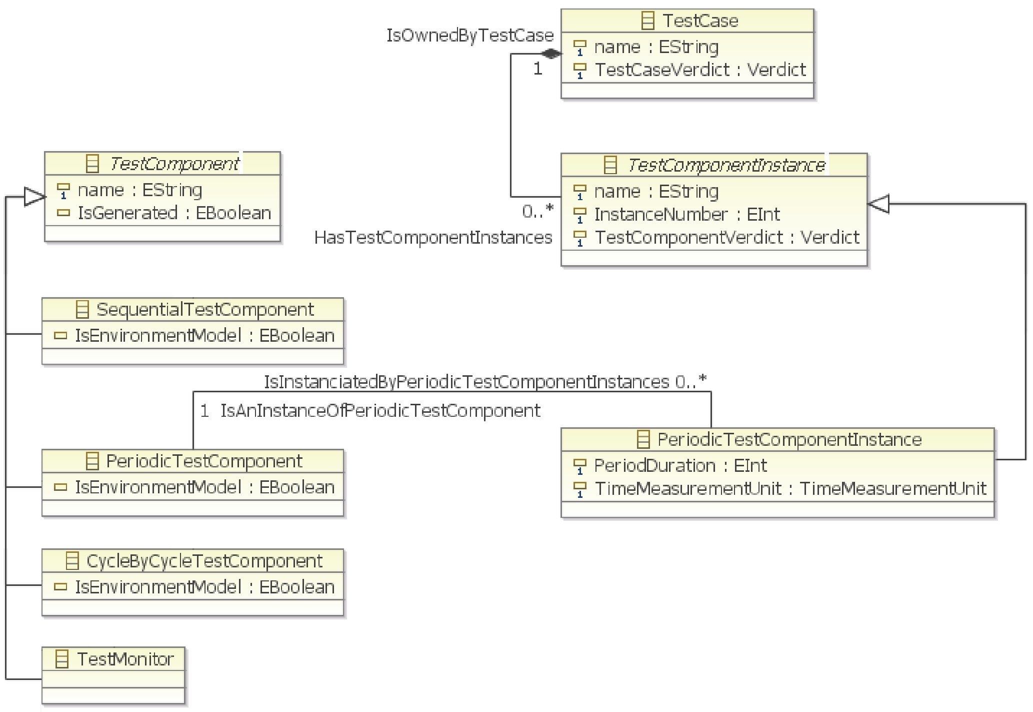

TestCase
name : EString
TestCaseVerdict : Verdict
IsOwnedByTestCase
1
0..*
HasTestComponentInstances
TestComponentInstance
name : EString
InstanceNumber : EInt
TestComponentVerdict : Verdict
TestComponent
name : EString
IsGenerated : EBoolean
SequentialTestComponent
IsEnvironmentModel : EBoolean
IsInstanciatedByPeriodicTestComponentInstances 0..*
1 IsAnInstanceOfPeriodicTestComponent
PeriodicTestComponent
IsEnvironmentModel : EBoolean
PeriodicTestComponentInstance
PeriodDuration : EInt
TimeMeasurementUnit : TimeMeasurementUnit
CycleByCycleTestComponent
IsEnvironmentModel : EBoolean
TestMonitor

**Figure 5.9: TestComponent Types Excerpt -**

A test component has a number of formal interfaces: `Parameter` and `Accessor` elements (Figure 5.10). They were defined in order to allow the reuse and multiple-instantiation of the test components.

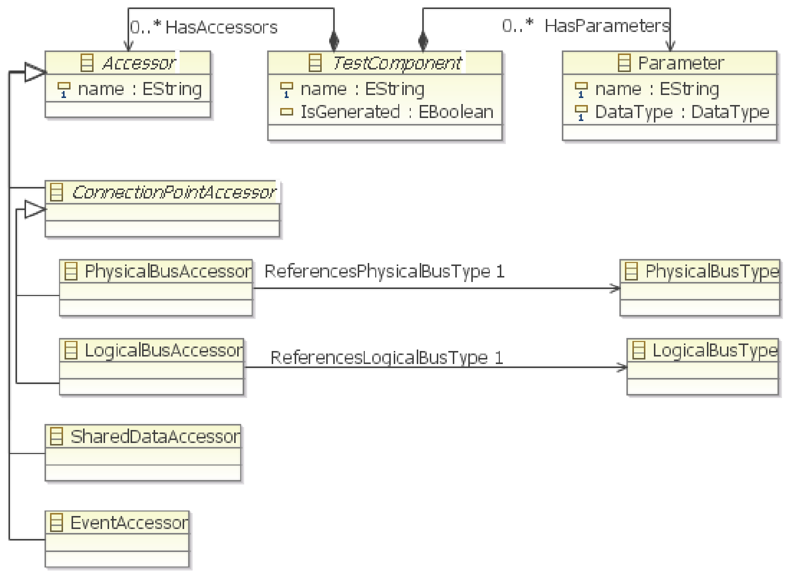


**Figure 5.10: TestComponent Interfaces Excerpt -**

A parameter is given a value when instantiating a test component (i.e., call-by-value argument). For example, if a test component verifies a behaviour of the SUT where the value of an application parameter is checked against several nominal values and tolerances, then the Nominal_Value and Tolerance can be parametrized. Not shown here, each test component instance owns `ParameterInitialization` elements that link a parameter to a ParameterInitialValue (the one given at instantiation). During the execution of the test component instance, the initial values of the parameters cannot be modified.

The `Accessor` elements are used in order to reuse a same test component on different: connection point instances, property instances, shared data or events (i.e., call-by-reference argument). Notice that the `ConnectionPointAccessor` elements are strongly-typed, as they reference `ConnectionPointType` elements. This applies also to the `PropertyAccessor`.

For example, if several application parameters need to be verified against their Nominal_Value and Tolerance, then an application parameter accessor should be defined. The behaviour of the test component would only manipulate the application parameter accessor. Afterwards, within a test case, a test architecture would link the application parameter accessor to the different application parameter instances, for each of its test component instances. An OCL rule (Listing 5.1) checks that the application parameter accessor and the application parameter instance are of the same type (i.e., refer the same `ApplicationParameterType`). For more information on the test architecture see Subsection 5.4.5.

Not all types of connection point accessor types are shown in Figure 5.10, but the test meta-model cover all ICD hierarchical levels:

- `ConnectorAccessor`,
- `PinAccessor`,
- `PhysicalBusAccessor`,
- `LogicalBusAccessor`,
- `MessageAccessor`,
- `MessageFieldAccessor`,
- `ApplicationParameterAccessor`.

In addition to the indirect access via its formal interfaces, a test component also has a direct access to all SUT interfaces. This direct access, although convenient, can lead to clashes when a same SUT interface is accessed by different test component instances. In order to alleviate such problems, we propose the definition of ownership relations between test component instances and SUT interfaces. The test architecture comprises the definition of these relations. OCL rules prescribe that only owner test component instances make side-effect accesses to owned SUT interfaces.

```
context ConnectionPointAccessorConnection
inv ApplicationParameterAccessorType_Is_ApplicationParameterInstanceType:
   self.ProvidesAccessForConnectionPointAccessor
  ->select(aAccessor:Accessor | aAccessor.oclIsTypeOf(
        ApplicationParameterAccessor)).oclAsType(
        ApplicationParameterAccessor).
   ReferencesApplicationParameterType.name
  = self.ProvidesAccessToConnectionPointInstance
  ->select(anInstance:ConnectionPointInstance | anInstance.oclIsTypeOf(
        ApplicationParameterInstance)).oclAsType(
        ApplicationParameterInstance).
   IsOfApplicationParameterType.name
```

**Listing 5.1:** OCL Rule Example - ApplicationParameterAccessor Type Verification

We would like to finally discuss the possibility to declare `ConnectionPointInstance-Alias` elements in order to shorten the identifiers of connection point instance elements (Figure 5.11). These aliases do not act as formal interfaces for test components. A test component can own several aliases, declared through the `HasConnectionPoint-InstanceAliases` EReference. Each alias is linked to a SUT interface through the `IsAliasForConnectionPointInstance` EReference.

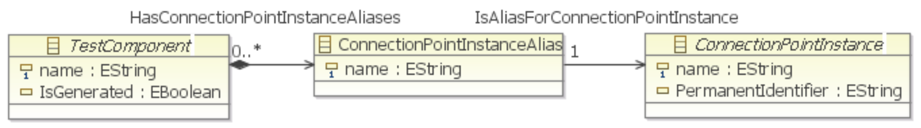

HasConnectionPointInstanceAliases
IsAliasForConnectionPointInstance
TestComponent
name : EString
IsGenerated : EBoolean
0..*
ConnectionPointInstanceAlias
name : EString
1
ConnectionPointInstance
name : EString
PermanentIdentifier : EString

**Figure 5.11: TestComponent - ConnectionPointInstanceAlias Excerpts -**

### 5.4.4 Verdict Management

Having presented the test case and test component/test component instance concepts, it is now appropriate to discuss the verdict management considered for the test meta-model. Notice in Figures 5.5 and 5.9 the `TestCaseVerdict` and `TestComponent-Verdict` EAttribute elements.

As the proprietary test languages in the sample we analysed (from $PL_1$ to $PL_4$) lacked rich verdict management functionalities, we decided to propose a solution inspired from the one proposed by TTCN-3 [83]. Our test meta-model defines five verdicts, with an order relation defined between them: `None` > `Pass` > `Inconclusive` > `Fail` > `Error`.

The verdict of a high-level container (i.e., test case) is automatically computed from the verdicts of the lower-level containers it includes (i.e., test component instances). For example, let us consider a test case that owns two test component instances. If the verdicts of the two test component instances are `Pass` and respectively `Fail`, then the verdict of the test case is `Fail` (the `Fail` verdict has a higher order position than the `Pass` verdict). A verdict can be overridden only by a verdict with a higher order position. For example, an `Inconclusive` verdict can only changed if a `Fail` or `Error` verdict arises. The `Error` verdict can only be set by the runtime/test platform when the execution of the test case and test component instances encounters problems, for example: when test actions are not executed or when the behaviour of a periodic/cycle-by-cycle test component does not hold its timing constraints.

### 5.4.5 TestArchitecture

Figure 5.12 presents the `TestArchitecture` concept. A test architecture is always associated with a test case.

The test architecture defines the connection of test component instance accessors to SUT interfaces, SUT interfaces properties, shared data and events. This is achieved through `Connection` elements. It also defines the ownership relation between test component instances and the same set of SUT interfaces, SUT interfaces properties, shared data and events. This is achieved through `OwnershipRelation` elements, that implement the policy of one producer and potentially many consumers that we chose.

For example, notice in Figure 5.12 how a `ConnectionPointAccessorConnection` links a SUT interface accessor to a SUT interface. It defines the connection using a triplet: the `TestComponentInstance` with its `ConnectionPointAccessor` and the targeted `ConnectionPointInstance`. Figure 5.12 also shows how a test component instance is declared the owner of an SUT interface. This capability is useful for clash detection, as only the owner is allowed to call test actions with side-effects (Figure 5.2) on the owned SUT interface. Clash detection can also be performed for access to SUT interface properties, shared data and events.

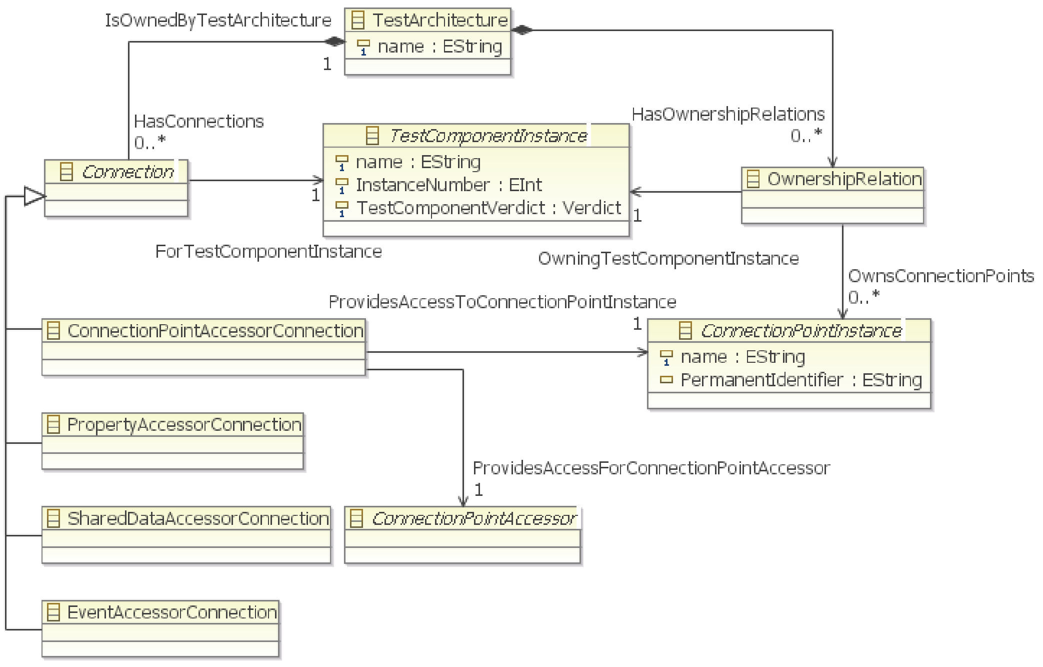


**Figure 5.12: TestArchitecture Excerpt -**

### 5.4.6 TestGroup and TestSuite

As mentioned previously when analysing the sample of test languages, inter-test organization is not a major issue for test languages. For this reason, we addressed in our test meta-model only basic functionalities of this type.

A `TestGroup` is used for the logical organization of test cases (Figure 5.13). This functionality is useful for requirements management. All test cases linked to a specific requirement on the behaviour of the SUT can be put together within the same test group

A `TestSuite` is used in order to define the execution order of test cases (Figure 5.13). A test suite comprises a tree-like structure of test suite nodes: an execution tree. Each test suite node corresponds to the execution of a test case, and has one parent (except for the first node) and a maximum of five children. These relations are defined through the `ReferencesParentTestSuiteNode` and `ReferencesChildTestSuiteNodes` EReference elements. Which of the five children is to be executed next depends of the verdict obtained following the execution of the test case: `None`, `Pass`, `Inconclusive`, `Fail`, `Error`. For this facility, notice the `ParentNodeVerdictCondition` EAttribute. For example, one can define a test suite where if the verdict of a first TestCase_1 `TestCase` is `Fail` then the TestCase_2 `TestCase` is executed next.

We would also like to introduce here the concept of state of the SUT (Figure 5.13). A `State` is defined by a *logicalCondition* on the SUT interfaces. For example, different Flight_Phase `State` elements can be defined for an Aircraft `SystemUnderTest`: Pre-Flight, Taxi, Take-Off, Departure, Climb, Cruise, Descent, Approach and Arrival. Each flight phase depends on the values of a number of application parameters, such as: Aircraft_Speed and Aircraft_Altitude. For example, the Cruise state is defined for an interval of the aforementioned application parameters: Aircraft_Speed $\in[Speed_{Min}, Speed_{Max}]$ and Aircraft_Altitude $\in[Altitude_{Min}, Altitude_{Max}]$. The definition of states for a SUT is useful as some test cases require the SUT to be in a specific state before executing (see the `RequiresSystemUnderTestState` EReference). If this is not the case, then an alert can be raised to the user with information on the incompatibility between the test case and the state of the SUT. In this case, the test case is not executed. This functionality was deemed interesting following discussions with test engineers.

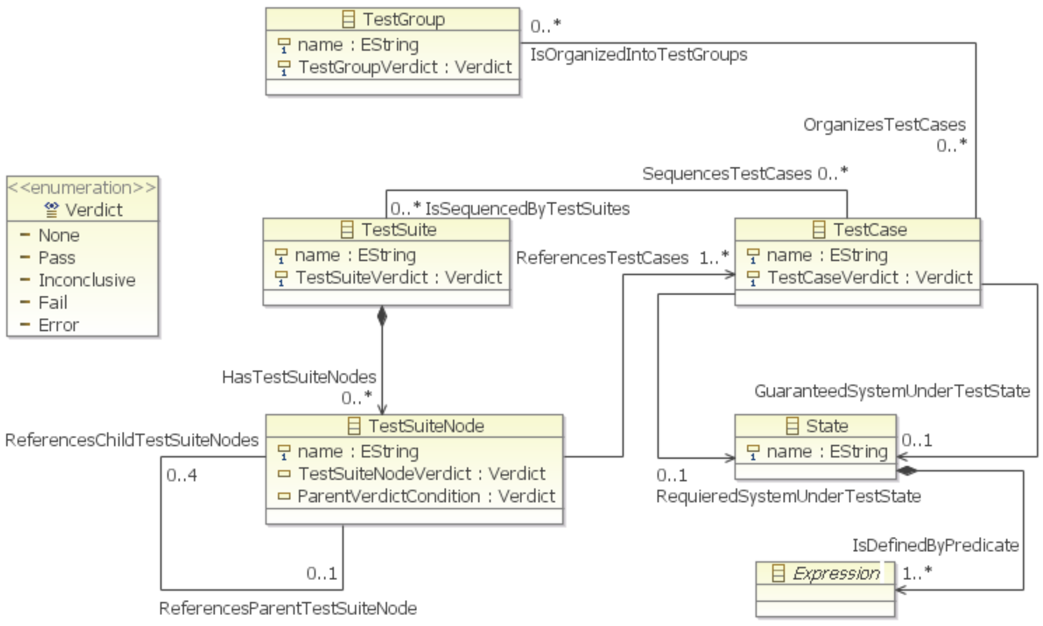
## 5.5 Low-Level Structural Concepts

All of the high-level structural concepts we presented previously (the test case and the test component types) have their behaviour organized into low-level structural concepts.

A test case organizes its behaviour inside `TestSection` containers (Figure 5.14). Test sections are executed sequentially. This allows test engineers to organise their test case, for example by defining an Initialization test section as well as a test section for each test objective that is being verified.

A test component organizes its behaviour inside `TestComponentElement` containers. Specializations exist for each type of test component, with the exception of the periodic test component (i.e., its behaviour usually lasts for a short cycle and is afterwards repeated): `TestMonitorElement`, `SequentialTestComponentElement` and `CycleBy-CycleTestComponentElement`.

A test monitor has a `TestMonitorPredicateElement` and a `TestMonitorContainer-Element` (Figure 5.15). The `TestMonitorPredicateElement` contains the *logicalCondition* that must become valid before the execution of the *behaviour* contained in the `Test-MonitorContainerElement`.



TestGroup
name : EString
TestGroupVerdict : Verdict
0..*
IsOrganizedIntoTestGroups
OrganizesTestCases
0..*
SequencesTestCases 0..*
0..* IsSequencedByTestSuites
<<enumeration>>
Verdict
None
Pass
Inconclusive
Fail
Error
TestSuite
name : EString
TestSuiteVerdict : Verdict
ReferencesTestCases 1..*
TestCase
name : EString
TestCaseVerdict : Verdict
HasTestSuiteNodes
0..*
GuaranteedSystemUnderTestState
ReferencesChildTestSuiteNodes
TestSuiteNode
name : EString
TestSuiteNodeVerdict : Verdict
ParentVerdictCondition : Verdict
0..4
State
name : EString
0..1
0..1
RequieredSystemUnderTestState
IsDefinedByPredicate
Expression
1..*
0..1
ReferencesParentTestSuiteNode

Figure 5.13: TestGroup and TestSuite Excerpt -

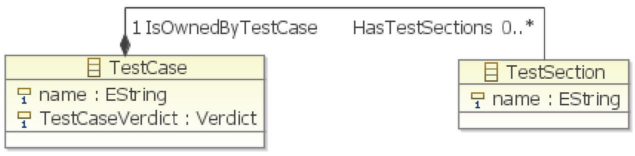


Figure 5.14: TestCase - TestSection Excerpt -

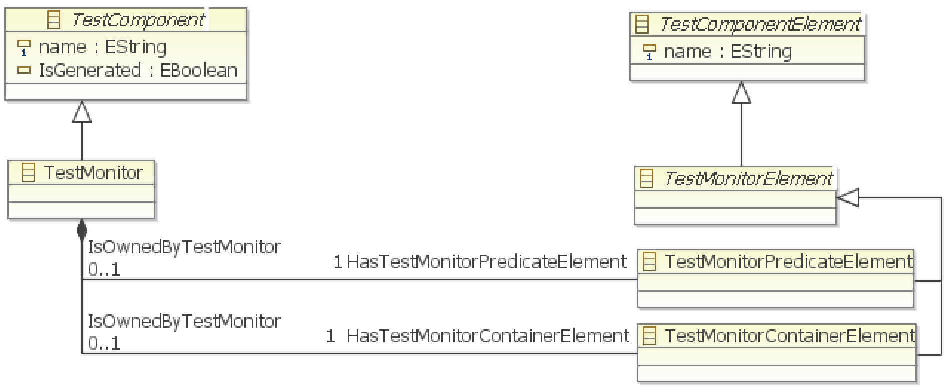


Figure 5.15: TestMonitor - TestMonitorElement -

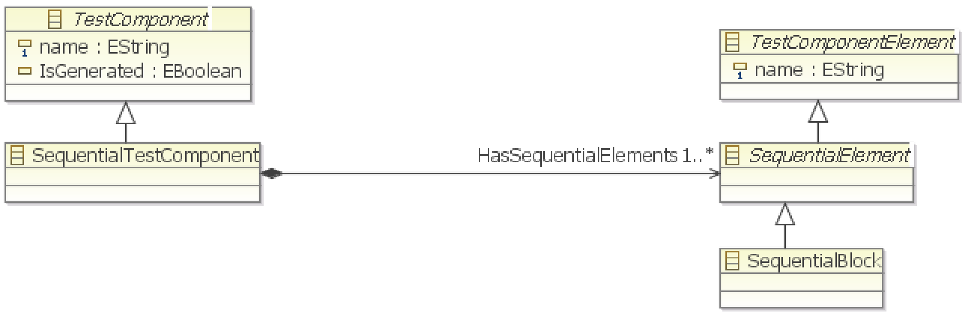


**Figure 5.16: SequentialTestComponent - SequentialTestComponentElement -**

A sequential test component has a set of `SequentialBlock` elements (Figure 5.16). Similarly to the test sections of a test case, sequential blocks are executed sequentially. They serve a similar purpose as the test sections, allowing test engineers to structure the behaviour of the sequential test component into meaningful blocks (e.g., initialization, stimulation, verification).

Finally, a cycle by cycle test component has a number of low-level structural concepts that allow a test engineer to specify different behaviours for each cycle or sequence of cycles: `Cycle`, `IteratedCycle` and `ConditionRepeatedCycle` (Figure 5.17). The `Cycle` has a behaviour to be executed only once, for a cycle. The `IteratedCycle` has a behaviour to be executed for a fixed number of sequential cycles (`NumberOfCycle-Iterations` EAttribute). Finally, the `ConditionRepeatedCycle` has a behaviour to be executed for a number of cycles dependant on the evaluation of a $logicalCondition$. The execution of a condition repeated cycle ends when the $logicalCondition$ becomes false or when the `MaximumIterations` EAttribute number is reached. By instantiating the elements described above several times, a test engineer can easily specify a complex behaviour, such as the one presented in Figure 5.18.

## 5.6 Behavioural Concepts

Each of the low-level structural concepts employed for the organization of the behaviour of test case and test component types owns a specific `Behavior`, depending on their type. Figure 5.19 shows the different low-level structural concepts on the left, with the corresponding behaviour on the right.

Each behaviour type is comprised of specific `Statement` elements. The allowed statements depend on the type of the element to which the behaviour is attached. This adds a layer of supervision over the test engineer's activity and is in favour of fault avoidance. As mentioned previously, the test components are in charge of the interaction with the SUT, being able to call test actions on its interfaces; while the

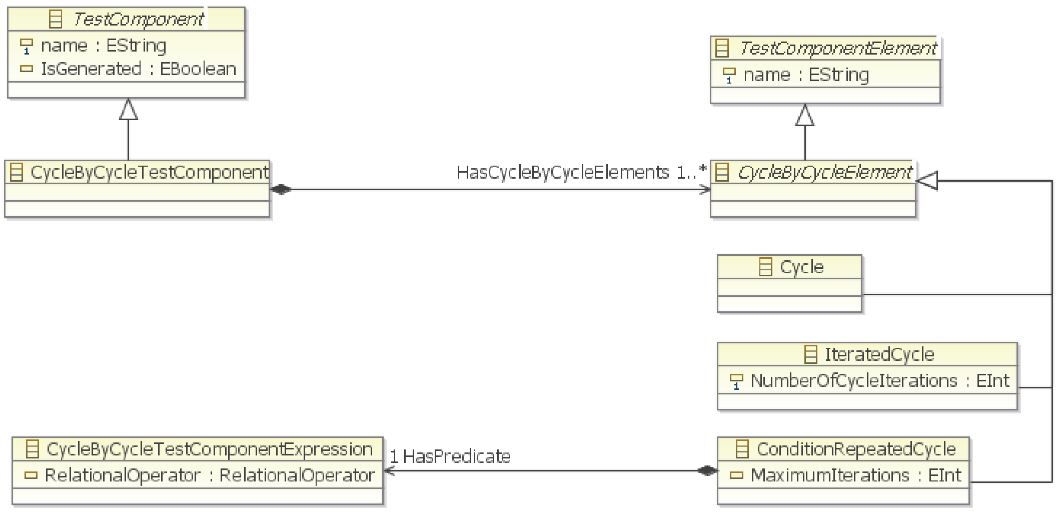
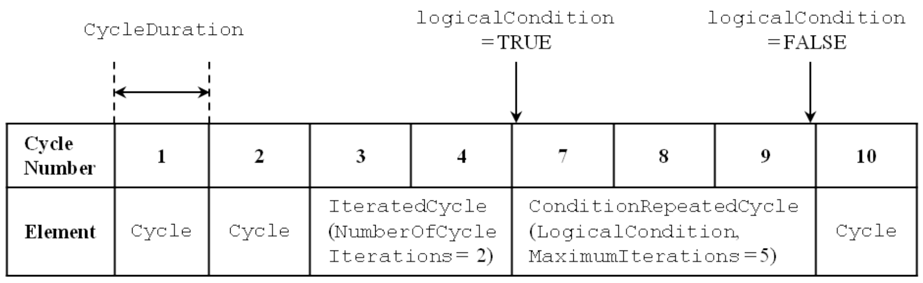


**Figure 5.17: CycleByCycleTestComponentTestComponent - CycleByCycleTestComponentElement -**



| Cycle Number | 1 | 2 | 3 | 4 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|
| Element | Cycle | Cycle | IteratedCycle (NumberOfCycle Iterations= 2) | | ConditionRepeatedCycle (LogicalCondition, MaximumIterations=5) | | | Cycle |

**Figure 5.18: CycleByCycleTestComponent - Complex Behaviour Example -**

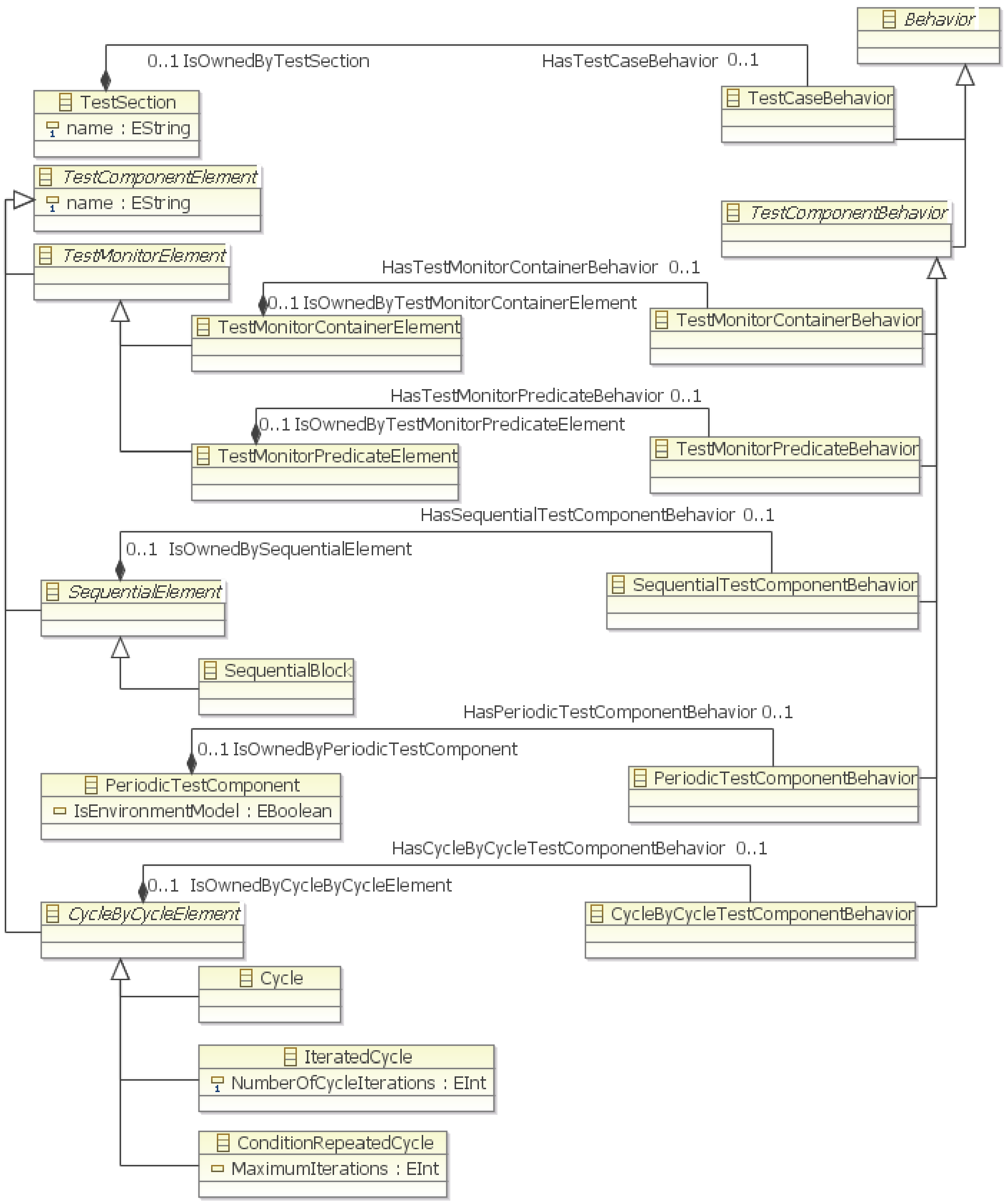

Behavior
TestSection
name : EString
TestCaseBehavior
0..1 IsOwnedByTestSection
HasTestCaseBehavior 0..1
TestComponentElement
name : EString
TestComponentBehavior
TestMonitorElement
HasTestMonitorContainerBehavior 0..1
0..1 IsOwnedByTestMonitorContainerElement
TestMonitorContainerElement
TestMonitorContainerBehavior
HasTestMonitorPredicateBehavior 0..1
0..1 IsOwnedByTestMonitorPredicateElement
TestMonitorPredicateElement
TestMonitorPredicateBehavior
HasSequentialTestComponentBehavior 0..1
0..1 IsOwnedBySequentialElement
SequentialElement
SequentialTestComponentBehavior
SequentialBlock
HasPeriodicTestComponentBehavior 0..1
0..1 IsOwnedByPeriodicTestComponent
PeriodicTestComponent
IsEnvironmentModel : EBoolean
PeriodicTestComponentBehavior
HasCycleByCycleTestComponentBehavior 0..1
0..1 IsOwnedByCycleByCycleElement
CycleByCycleElement
CycleByCycleTestComponentBehavior
Cycle
IteratedCycle
NumberOfCycleIterations : EInt
ConditionRepeatedCycle
MaximumIterations : EInt

**Figure 5.19: Behavior Excerpt -**

test case is in charge of controlling the execution of the corresponding test component instances. The relation between the behaviour and corresponding accessible statements is defined on their specializations, for example: `TestCaseBehavior` and `TestCase-Statement` (Figure 5.20). This was necessary in order to prohibit, for example, a `Test-ComponentBehavior` to access `TestCaseStatement` elements, which would have become accessible because of the Inheritance relation, had it been defined on the generalizations: `Behavior` and `Statement`.

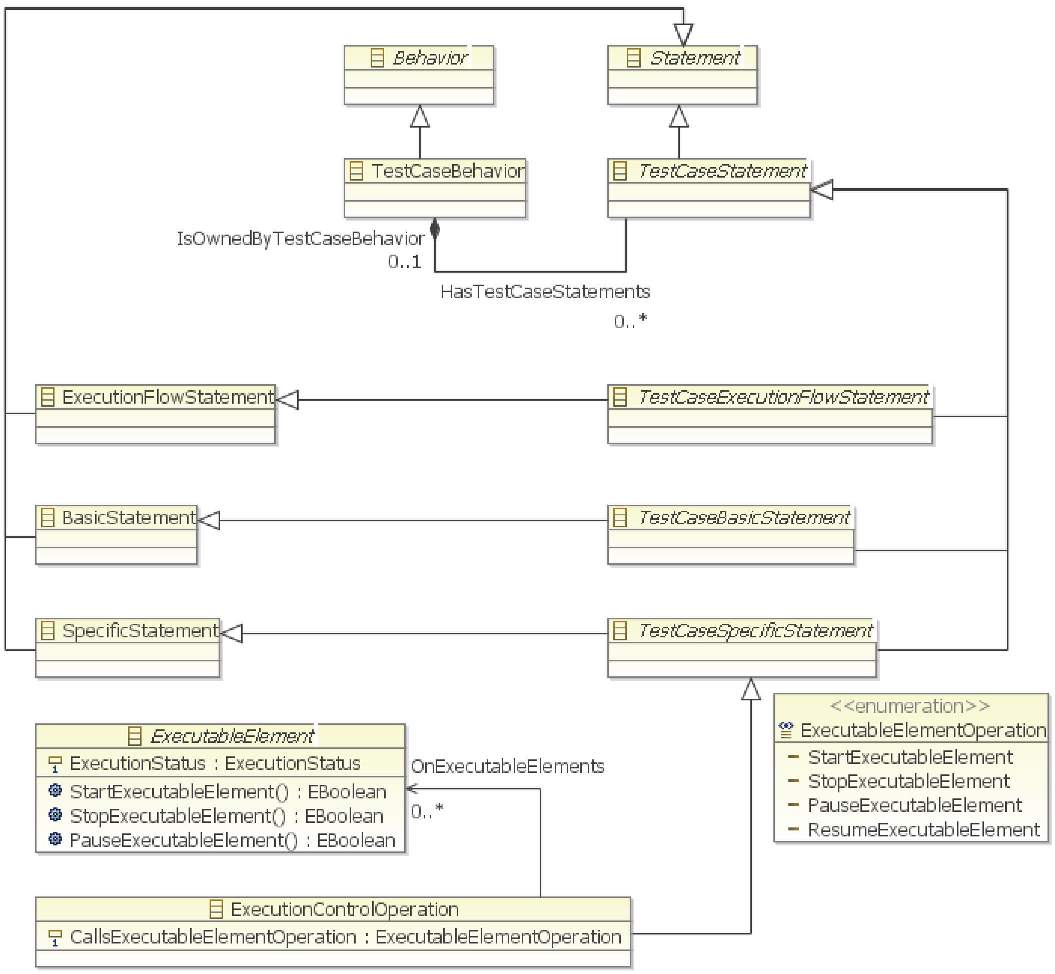

Behavior
Statement
TestCaseBehavior
TestCaseStatement
IsOwnedByTestCaseBehavior
0..1
HasTestCaseStatements
0..*
ExecutionFlowStatement
TestCaseExecutionFlowStatement
BasicStatement
TestCaseBasicStatement
SpecificStatement
TestCaseSpecificStatement
ExecutableElement
ExecutionStatus : ExecutionStatus
StartExecutableElement() : EBoolean
StopExecutableElement() : EBoolean
PauseExecutableElement() : EBoolean
OnExecutableElements
0..*
<<enumeration>>
ExecutableElementOperation
StartExecutableElement
StopExecutableElement
PauseExecutableElement
ResumeExecutableElement
ExecutionControlOperation
CallsExecutableElementOperation : ExecutableElementOperation

**Figure 5.20: TestCaseStatement Excerpt -**

Three types of statements have been defined, with each type being specialized itself for each of the different high-level structural concepts (Figures 5.20 and 5.21):

- `ExecutionFlowStatement` (Subsection 5.6.1),

- `BasicStatement` (Subsection 5.6.2),
- `SpecificStatement` (Subsection 5.6.3).

### 5.6.1 ExecutionFlowStatement

An `ExecutionFlowStatement` is used for the definition of instructions that control the execution flow, for example:

- `ConditionalStatement` (i.e., if ... then ... else),
- `IterationStatement` (i.e., for ... ),
- `RepetitionStatement` (i.e., while ... ).

Observe the three sequential test component execution flow statements in Figure 5.21: `STCConditionalStatement`, `STCIterationStatement` and `STCRepetitionStatement`. Each one of these elements can own one or two sequential test component behaviours. The `STCConditionStatement` owns two, one mandatory in case the conditional predicate (not shown) evaluates to true (`HasConsequentBehavior` EReference), and one optional in case it evaluates to false (`HasAlternativeBehavior` EReference). Notice the optional `MaximumIterations` EAttribute of the `STCRepetitionStatement`. In order to contribute to fault avoidance, this EAttribute is rendered mandatory in the case of a periodic and cycle-by-cycle test component.

### 5.6.2 BasicStatement

A `BasicStatement` is used for generic instructions, for example:

- `VariableDeclarationStatement` (i.e., int variableName),
- `VariableAssignmentStatement` (i.e., variableName = newValue).

We also briefly discuss in this subsection the expressions and literals used in the statements. For example, in Figure 5.22, the conditional statement has a logical expression (`HasPredicate` EReference) and the action call statement has expressions passed as parameters. We will not go into further detail here concerning the meta-modelling of expressions. It suffices to say that we cover a rich set of possible expressions, such as:

- logical expressions: `And`/`Or`,
- equality expressions: `EqualTo`/`NotEqualTo`,
- comparison expressions: `GreaterThan`/`GreaterThanOrEqualTo` and `LessThan`/`LessThanOrEqualTo`,

- multiplication expressions: `Multiplication/Power/Division/Modulo`,
- addition expressions: `Plus/Minus`,
- unary expressions: `Plus/Minus/Negation`.

We also defined four types of literals that can be used inside the expressions: `BooleanLiteral`, `IntegerLiteral`, `FloatLiteral` and `StringLiteral`. For more information on the topic of expressions and data types, please see the following two examples: the Arithmetics tutorial that accompanies Xtext [123] and the partial programming language Xbase (defined with Xtext). When defining the test meta-model we took inspiration from these two, as well as from the grammars of known general-purpose programming languages (e.g., C++, Python).

### 5.6.3 SpecificStatement

A `SpecificStatement` is used for instructions that are specific to the high-level structural concept, for example:

- `ExecutionControlStatement` for a `TestCase` (Figure 5.20),
- `STCTestActionCallStatement` for a `TestComponent` (we prefix all statements regarding a sequential test component with STC, Figure 5.21).

The `ExecutionControlStatement` is used to start/stop/pause/resume (`ExecutableElementOperation EEnum`) the execution of a test component instance by the test case.

A specific statement of test components is the `TestActionCallStatement`. It is used to call test actions on SUT interfaces, toolkits and accessors. Notice the `STCTestActionCallStatement` in Figure 5.21. Also observe its different EAttribute elements that are linked to the type of test action that is being called (Figure 5.2). For example, a `Timed::TestAction` has to be called with a specific a `DurationValue`, while a `Bounded::TestAction` can be called with a specific `MaximumDuration`. As mentioned previously, this information is in favour of fault avoidance as the behaviour can be analysed before its execution. For example, we were able to identify by means of OCL rules an incorrect specification where test actions were called with a duration higher than the period of a periodic test component. A simplified rule can be found in Listing 5.2. We also defined a more complex version that covers the imbrication of statements contained by other statements. We achieved its definition by using the transitive closure offered by OCL, with which recursion can be expressed.

The `STCTestConditionEvaluationStatement` is used in order to modify the verdict of a sequential test component. It verifies a *logicalCondition* and sets the verdict accordingly. For example, if the *logicalCondition* evaluates to false and the previous verdict was `Pass`, then the verdict becomes `Fail`.

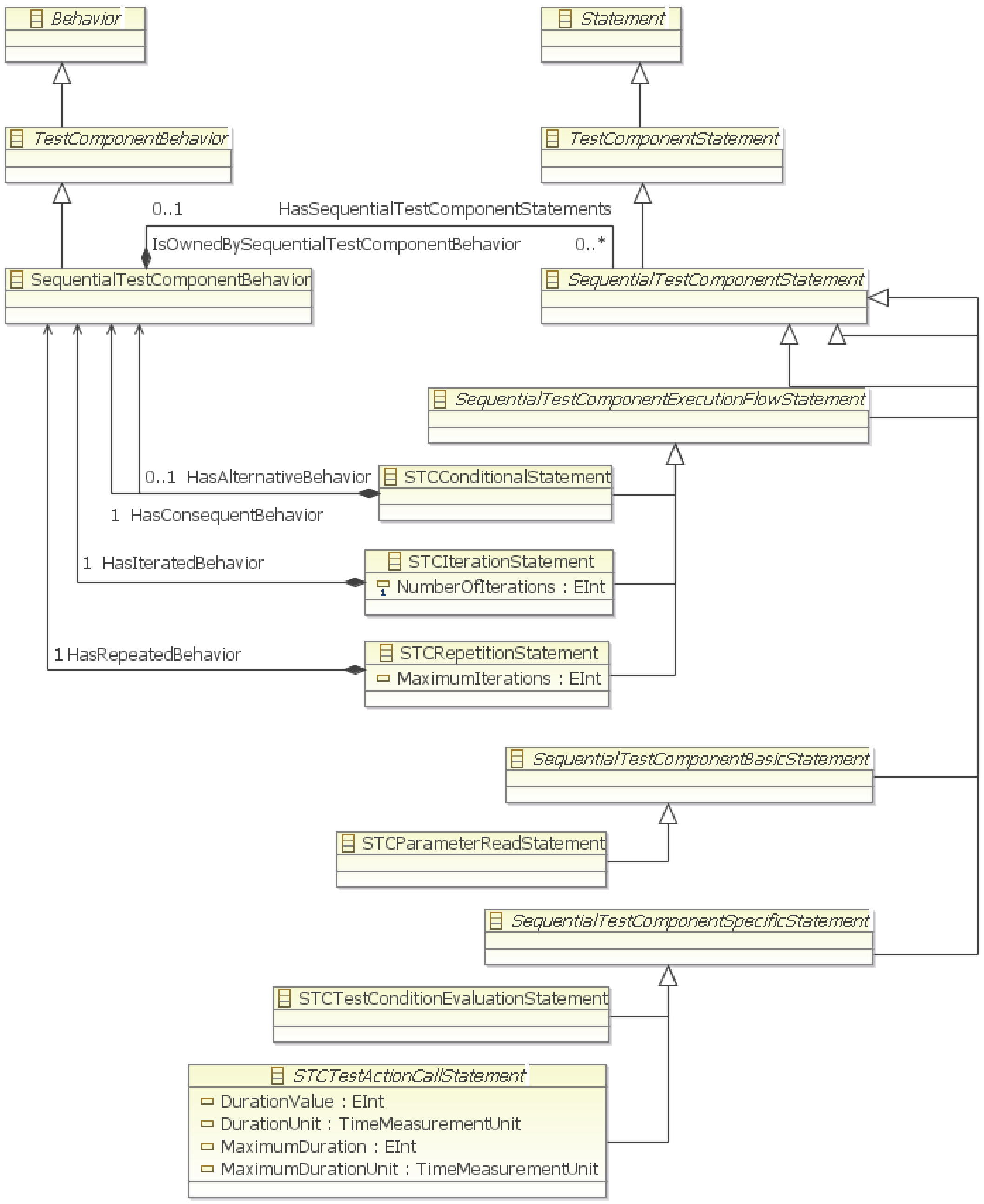
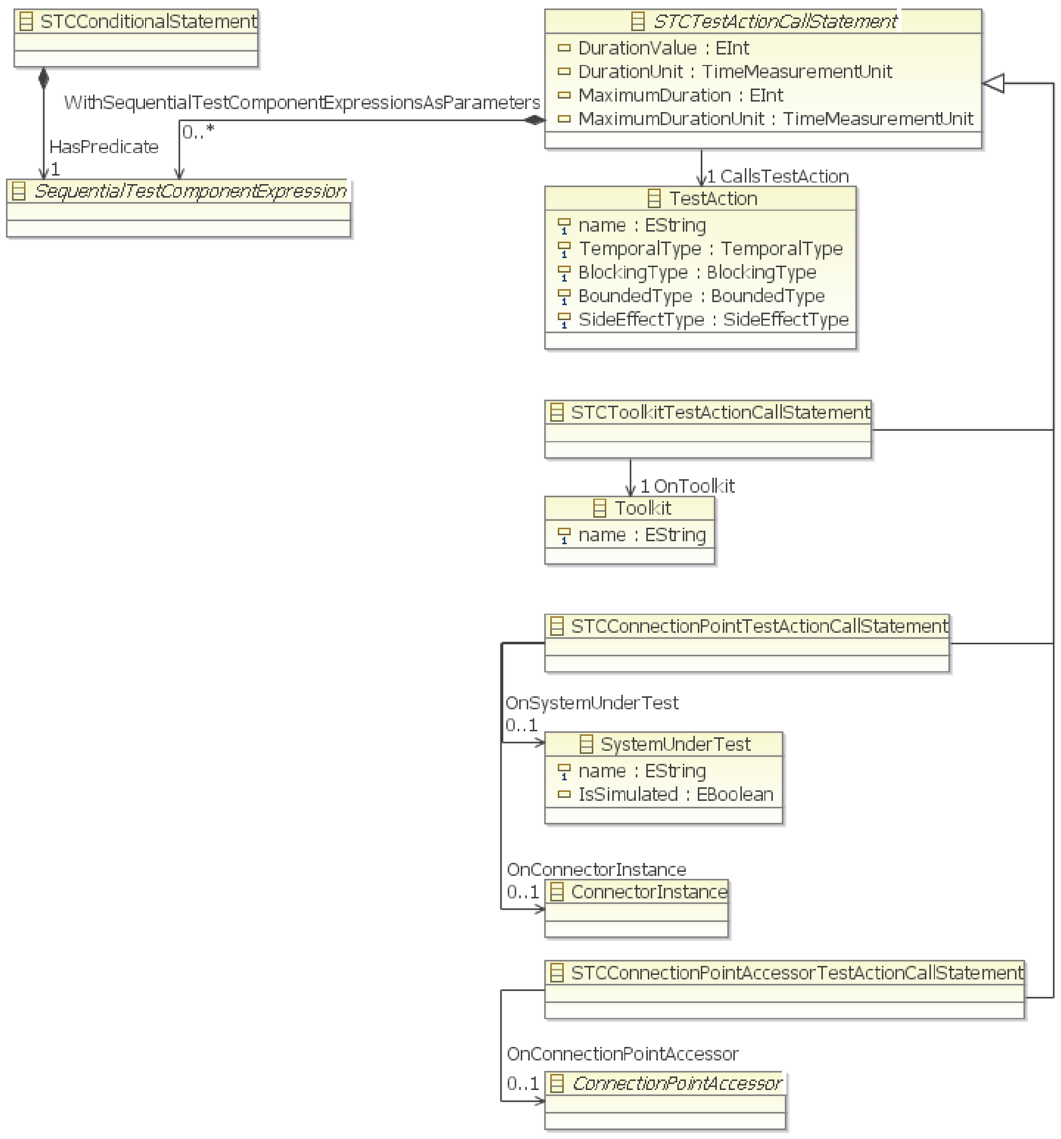

Behavior
Statement
TestComponentBehavior
TestComponentStatement
0..1
HasSequentialTestComponentStatements
IsOwnedBySequentialTestComponentBehavior
0..*
SequentialTestComponentBehavior
SequentialTestComponentStatement
SequentialTestComponentExecutionFlowStatement
0..1 HasAlternativeBehavior
STCConditionalStatement
1 HasConsequentBehavior
1 HasIteratedBehavior
STCIterationStatement
NumberOfIterations : EInt
1 HasRepeatedBehavior
STCRepetitionStatement
MaximumIterations : EInt
SequentialTestComponentBasicStatement
STCParameterReadStatement
SequentialTestComponentSpecificStatement
STCTestConditionEvaluationStatement
STCTestActionCallStatement
DurationValue : EInt
DurationUnit : TimeMeasurementUnit
MaximumDuration : EInt
MaximumDurationUnit : TimeMeasurementUnit

**Figure 5.21: SequentialTestComponentStatement Excerpt -**



STCConditionalStatement
STCTestActionCallStatement
DurationValue : EInt
DurationUnit : TimeMeasurementUnit
MaximumDuration : EInt
MaximumDurationUnit : TimeMeasurementUnit
WithSequentialTestComponentExpressionsAsParameters
0..*
HasPredicate
1
SequentialTestComponentExpression
1 CallsTestAction
TestAction
name : EString
TemporalType : TemporalType
BlockingType : BlockingType
BoundedType : BoundedType
SideEffectType : SideEffectType
STCToolkitTestActionCallStatement
1 OnToolkit
Toolkit
name : EString
STCConnectionPointTestActionCallStatement
OnSystemUnderTest
0..1
SystemUnderTest
name : EString
IsSimulated : EBoolean
OnConnectorInstance
0..1
ConnectorInstance
STCConnectionPointAccessorTestActionCallStatement
OnConnectionPointAccessor
0..1
ConnectionPointAccessor

**Figure 5.22: SequentialTestComponent - Details Example -**

```
context CycleByCycleTestComponentInstance
inv TestActionCallStatement_DurationValue_Smaller_Than_CycleDuration:
  self.IsAnInstanceOfCycleByCycleTestComponent.HasCycleByCycleElements.
  HasCycleByCycleTestComponentBehavior.
      HasCycleByCycleTestComponentStatements
  ->select(aStatement : CycleByCycleTestComponentStatement | aStatement.
      oclIsKindOf(CBCTCTestActionCallStatement)).
  oclAsType(CBCTCTestActionCallStatement)
  ->select(aStatement : CBCTCTestActionCallStatement | aStatement.
      CallsTestAction.TemporalType = TemporalType::Timed)
  ->forAll(DurationValue <= self.CycleDuration)
```

**Listing 5.2:** OCL Rule Example - Behavioural Concepts

In addition to the different statements discussed here, the test meta-model also contains the following ones, used for manipulation of events defined in the test context: `EventWaitStatement`, `EventRaiseStatement`, `EventLowerStatement`. Manipulation of shared data is possible through the `ReadSharedData` and `WriteSharedData` instructions.

We have now finished the presentation of our test meta-model. We continue with a discussion on the graphical and textual editors that can be attached to the test meta-model in Section 5.7.

## 5.7 Test Model Development Environment

The Eclipse Modeling Framework (EMF) [119] offers us access to a wide range of tools, as mentioned at the beginning of this chapter. It is able to automatically generate a graphical editor of models from a meta-model. This graphical editor was used for the high/low-level structural concepts of the test meta-model. For the behavioural concepts, a textual editor was developed with Xtext [123]. These two editors (graphical and textual) were enriched and integrated inside the Man-Machine Interface (MMI) component of the U-TEST Real-Time System with the help of three participants: Gilles BALLANGER, Guilhem BONNAFOUS, Mathieu GARCIA and Etienne ALLOGO, at Cassidian Test & Services. Snapshots of our prototype can be found in Chapter 7.

### 5.7.1 Graphical Editor

This graphical editor presents a tree-like structure. Figure 5.23 shows a snapshot of the automatically generated graphical editor. It is used only for the instantiation of EReference elements that have their Containment set to true (e.g., the MyTestContext `TestContext` contains the MyTestCase `TestCase` and as such MyTestCase is a branch of MyTestContext). It is accompanied by a Properties view, which allows the modification of the rest of EReference elements, together with the EAttribute elements.

Contextual menus are offered. Although the automatically generated graphical editor covers all of the test meta-model elements, it can be modified in order to present only a subset of the test meta-model. In our prototype, some coding was necessary to produce two graphical editors: one for the `UserData` section, and one for the `ProviderData` section of the test meta-model; and to hide all child elements of `Behavior` specializations (Figure 5.19). The `UserData` and `ProviderData` graphical editors only show high/low-level structural concepts. Behavioural concepts are dealt with within the textual editor. This graphical editor is only one example of an interface attached to the test meta-model. For the purpose of our prototype, we considered this type of graphical editor to suffice. For example, GMF [121] and Graphiti [122] offer the possibility to design even more ergonomic and complex graphical editors.

### 5.7.2 Textual Editor

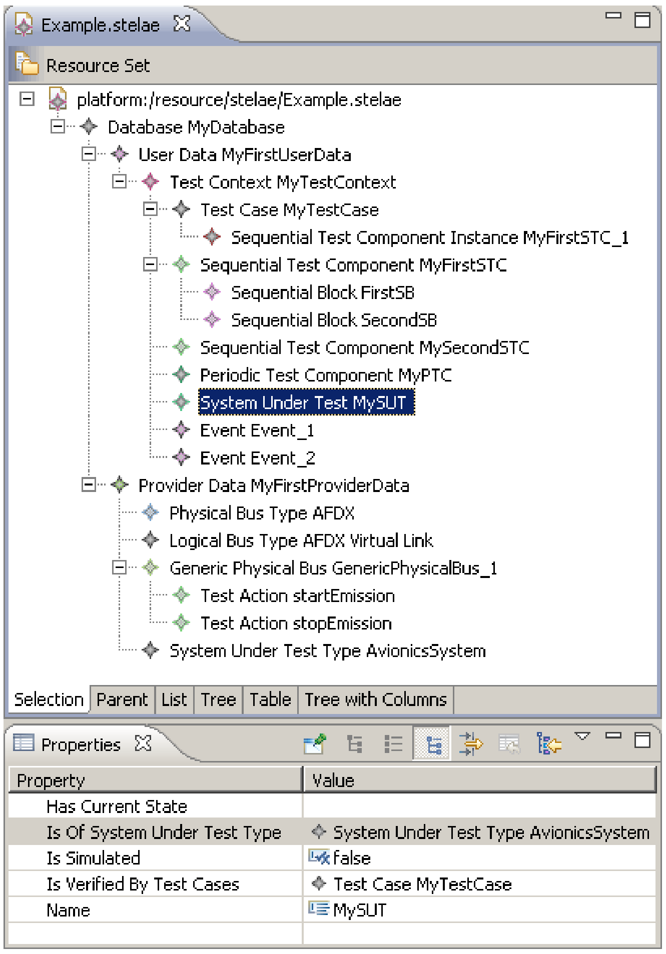
Xtext [123] is an open-source framework for developing programming languages and domain-specific languages (DSLs). Unlike standard parser generators, Xtext not only generates a parser, but also a class model for the Abstract Syntax Tree and a fully featured, customizable Eclipse-based IDE. To specify a language, a user has to write a grammar in Xtext's grammar language. This grammar describes how an Ecore model is derived from a textual notation. From that definition, a code generator derives an ANTLR parser and the classes for the object model. In addition, an Eclipse-based Integrated Development Environment (IDE) integration is generated. That IDE offers, among other functionalities: syntax colouring and code completion.

Similarly to traditional grammars that employ a Backus–Naur Form notation, an Xtext grammar consists of a number of derivation rules. Each derivation rule is a piece of static text with gaps, where each gap in the text can be filled with model information. The static text is always inserted between quotation marks.

Listing 5.3 shows a simplified grammar example, relative to the `STCToolkitTestActionCallStatement` that was discussed previously (Figure 5.22). Lines 1-7 present the grammar rule. Line 11 shows some code that conforms to the grammar rule. Line 15 present a different concrete syntax that could be defined in a similar manner as the first example, in order to illustrate the capacity for customisation of our approach.

The application of the rule to parse the compliant line of code yields the following result:

- the `Toolkit` name is *userCommunication* and `OnToolkit` is a reference to this model element,
- similarly, the called `TestAction` refers to *showMessage*,



```
platform:/resource/stelae/Example.stelae
  Database MyDatabase
    User Data MyFirstUserData
      Test Context MyTestContext
        Test Case MyTestCase
          Sequential Test Component Instance MyFirstSTC_1
        Sequential Test Component MyFirstSTC
          Sequential Block FirstSB
          Sequential Block SecondSB
        Sequential Test Component MySecondSTC
        Periodic Test Component MyPTC
        System Under Test MySUT
        Event Event_1
        Event Event_2
    Provider Data MyFirstProviderData
      Physical Bus Type AFDX
      Logical Bus Type AFDX Virtual Link
      Generic Physical Bus GenericPhysicalBus_1
        Test Action startEmission
        Test Action stopEmission
      System Under Test Type AvionicsSystem
```

| Property | Value |
|---|---|
| Has Current State | |
| Is Of System Under Test Type | System Under Test Type AvionicsSystem |
| Is Simulated | false |
| Is Verified By Test Cases | Test Case MyTestCase |
| Name | MySUT |

Figure 5.23: Default Graphical Editor Snapshot -

- a single parameter is identified between parentheses, the string expression 'Hello world!'.

Notice that the syntax of Line 11 adopts an object-oriented paradigm, where the showMessage `TestAction` acts as a method of the userCommunication `Toolkit` object. As mentioned previously, this is only one of the many concrete syntaxes/textual representations that can be attached to the test meta-model. The second example of definable concrete syntax (Line 15), although resembling the string solution offered by existing test languages, exhibits the same fault avoidance capabilities as the first example. The fact that the name of the test action appears within quotation marks is only a visual representation, as the test action remains a strongly-typed element of the test model.

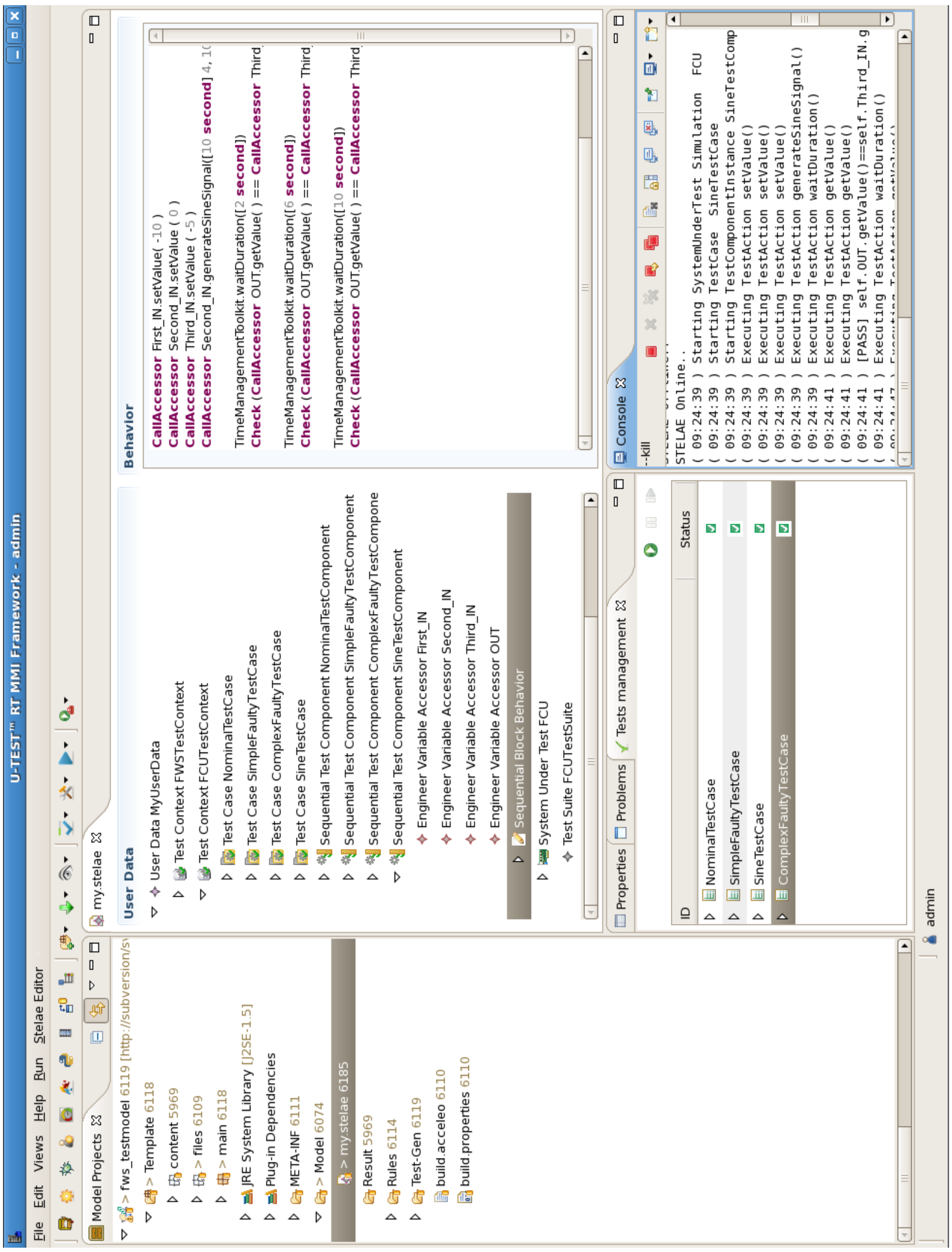
Although simple versions of the different graphical and textual editor functionalities are available out-of-the-box, some coding is necessary in order to implement semantics that is expressed within the meta-model but that is not automatically exploitable by the framework. For example, when a `STCToolkitTestActionCallStatement` calls a `TestAction` on a `Toolkit`, it also allows calling `TestAction` elements on the wrong `Toolkit`. Although the test meta-model contains information on which `TestAction` is attached to which `Toolkit` (the `HasProviderDefinedTestActions` EReference in Figure 5.2), this information is not directly exploitable when automatically generating the textual editor. The same holds true for the graphical editor in similar situations. As such, code has been developed in order to implement this semantics, for example in order for the code completion functionality to offer for a given `Toolkit` only the list of `TestAction` elements it comprises.

```
/* Derivation rule definition */

STCToolkitTestActionCallStatement returns
    STCToolkitTestActionCallStatement:

/* Derivation rule body */

OnToolkit = [Toolkit|EString] "." CallsTestAction = [TestAction|EString] "
    (" (WithSequentialTestComponentExpressionAsParameters+=
    SequentialTestComponentExpression ("," 
    WithSequentialTestComponentExpressionAsParameters+=
    SequentialTestComponentExpression)*)? ")"

/* Compliant line of code */

userCommunication.showMessage("Hello world!");

/* A different concrete syntax that could be defined */

userCommunication("showMessage","Hello world!");
```

**Listing 5.3:** Xtext STCToolkitTestActionCallStatement Excerpt

Figure 5.24 shows a screen-shot of the prototype mixed (graphical and textual) test model development environment, integrated into the Man-Machine Interface (MMI) component of the U-TEST Real-Time System [3] integration test platform developed by Cassidian Test & Services. Notice in the centre of the image the `UserData` section graphical editor, with the textual editor being visible on the right. When the test solution user selects in the graphical editor an element that possesses a `Behavior`, then automatically the textual editor opens, showing the corresponding `Behavior`. The `ProviderData` section graphical editor (not shown) is protected from unauthorized access, as only test solution provider users can modify this section.

In Chapter 7 we present a case study, where we define in the test model development environment a number of test models.


U-TEST™ RT MMI Framework - admin

File Edit Views Help Run Stelae Editor

Model Projects

- > fws_testmodel 6119 [http://subversion/s
  - > Template 6118
    - content 5969
    - > files 6109
    - > main 6118
  - JRE System Library [J2SE-1.5]
  - Plug-in Dependencies
  - META-INF 6111
  - > Model 6074
    - > my.stelae 6185
  - Result 5969
  - Rules 6114
  - Test-Gen 6119
  - build.acceleo 6110
  - build.properties 6110

my.stelae

User Data

- User Data MyUserData
  - Test Context FWSTestContext
  - Test Context FCUTestContext
  - Test Case NominalTestCase
  - Test Case SimpleFaultyTestCase
  - Test Case ComplexFaultyTestCase
  - Test Case SineTestCase
  - Sequential Test Component NominalTestComponent
  - Sequential Test Component SimpleFaultyTestComponent
  - Sequential Test Component ComplexFaultyTestCompone
  - Sequential Test Component SineTestComponent
    - Engineer Variable Accessor First_IN
    - Engineer Variable Accessor Second_IN
    - Engineer Variable Accessor Third_IN
    - Engineer Variable Accessor OUT
    - Sequential Block Behavior
  - System Under Test FCU
  - Test Suite FCUTestSuite

Behavior

```
CallAccessor First_IN.setValue( -10 )
CallAccessor Second_IN.setValue ( 0 )
CallAccessor Third_IN.setValue ( -5 )
CallAccessor Second_IN.generateSineSignal([10 second] 4, 1(

TimeManagementToolkit.waitDuration([2 second])
Check (CallAccessor OUT.getValue( ) == CallAccessor Third_

TimeManagementToolkit.waitDuration([6 second])
Check (CallAccessor OUT.getValue( ) == CallAccessor Third_

TimeManagementToolkit.waitDuration([10 second])
Check (CallAccessor OUT.getValue( ) == CallAccessor Third_
```

Properties  Problems  Tests management

| ID | Status |
|---|---|
| NominalTestCase | ✓ |
| SimpleFaultyTestCase | ✓ |
| SineTestCase | ✓ |
| ComplexFaultyTestCase | ✓ |

Console

```
--kill
STELAE Online.
( 09:24:39 ) Starting SystemUnderTest Simulation  FCU
( 09:24:39 ) Starting TestCase  SineTestCase
( 09:24:39 ) Starting TestComponentInstance SineTestComp
( 09:24:39 ) Executing TestAction setValue()
( 09:24:39 ) Executing TestAction setValue()
( 09:24:39 ) Executing TestAction setValue()
( 09:24:39 ) Executing TestAction generateSineSignal()
( 09:24:39 ) Executing TestAction waitDuration()
( 09:24:41 ) Executing TestAction getValue()
( 09:24:41 ) Executing TestAction getValue()
( 09:24:41 ) [PASS] self.OUT.getValue()==self.Third_IN.g
( 09:24:41 ) Executing TestAction waitDuration()
```

admin

Figure 5.24: Mixed (Graphical and Textual) Test Model Development Environment Screenshot -

## 5.8 Conclusion

In this chapter we have presented the test meta-model we defined. The definition represented an effort of six person-months (with no prior experience of the author with meta-modelling and EMF Ecore).

One of the challenges was to homogeneously integrate all of the domain-specific concepts we had previously identified. We achieved this objective, but the resulting test meta-model is inherently complex. For future industrialization purposes, this complexity could be hidden by developing wizards to guide the test engineer and automatically fill-in some test model elements.

We noticed that some constraints on the activity of test engineers with test models could be either introduced inside the test meta-model by construction, or added later by means of OCL [124] rules for example. A choice had to be made where part of the complexity of the approach is to be inserted. We preferred enriching the test meta-model, as we believe that not allowing a test engineer to perform an incorrect action is preferable to her/him doing it, verifying its correctness and afterwards correcting it. In addition, our approach tried to insert as much as possible of the domain-specific concepts and their relations inside the test meta-model, instead of having this information scattered around in the test meta-model, OCL rules, Java classes analysing test models, informal documents as well as inside the textual representation.

Another issue we encountered concerned the two levels used in model-driven engineering: the meta-model and model levels with one instantiation step between them. In most applications these two levels suffice, but in our work we encountered cases when three levels are required. In such cases, we employed the Type Object design pattern [127]. Although this pattern offers remarkable flexibility to create new components and connection points at the model level, its usage was one of the factors that led to the growth in complexity of the test meta-model.

Concerning OCL rules, we found them very practical in order to express constraints on the test models. However, they quickly grow in complexity. This is also due to the complexity of the test meta-model. Moreover, we found it difficult to express some behaviours with OCL. For example, we were interested in calculating the sum of all the durations with which test actions are called, in order to compare this result with the period of a periodic test component. In order to calculate the sum, the different durations would have to be converted to a same time measurement unit. We were not able to express this conversion directly in OCL. However, it is possible to call external operations coded in Java from OCL, that would do the necessary conversion.

The development of the graphical and textual editors for our demonstrator was very fast (the equivalent of a three person-months effort), as our first prototype required only basic functionalities. In their current state, the editors do not yet offer test engineers all the functionalities/short-cuts they would need. The prototype is called STELAE (Systems TEst LAnguage Environment). Our evaluation of the development of richer, more ergonomic editors, with technologies such as Graphical Modeling Framework (GMF) [121] or Graphiti [122], leads us to believe that an industrial product would require a much greater effort than that for our first prototype. One challenging issue we encountered when developing the two editors was to ensure their synchronization. The current state of the technology is not optimized for a usage of several editors synchronized on a same model.

Finally, we would like to mention that some activities performed manually by test engineers on test models should be automated in the future: such as the definition of the interfaces of the SUT that can be automated by parsing the ICD document.

# 6

# Test Model Implementation

In this chapter we illustrate the manner in which test models (conforming to the test meta-model that was previously discussed) are implemented by template-based model-to-text transformations. For this task, the Acceleo model-to-text tool [125] was chosen because of its tight integration within EMF [119]. The target test language is based on the Python programming language and was developed by Cassidian Test & Services. Inside the company, this test language is known as Automatic Test Language (ATL). In order to avoid confusion with the Atlas Transformation Language (also abbreviated ATL) [128], we shall refer to our test language as $PL_5$ in the rest of this document. $PL_5$ is executed on top of the U-TEST Real-Time System [3] integration test platform. $PL_5$ was not part of the sample of test languages analysed in Chapter 4, as it did not exist at that time. It allowed us to challenge the genericity of our test meta-model.

Our implementation covers only a subset of the test meta-model concepts, that were considered to be sufficient with regards to the case studies (Chapter 7). We followed a methodology to cover concepts that were offered completely, partially, or not offered at all by $PL_5$, in order to investigate the specificities and difficulties of automatic code generation in these cases. We also paid attention to whether the implementation part of our work would impact or not the test languages analysis or the test meta-model.

Section 6.1 presents the functionalities of the Acceleo tool on a simple meta-model example. This example captures, at a smaller scale, the main choices that were made for the implementation of the test meta-model.

Section 6.2 presents the Python-based target executable test language $PL_5$. We focus on the functionalities that it totally or partially offers, or does not offer at all, in comparison with the functionalities we defined in the test meta-model. We discuss five high-level feature categories: test case (Subsection 6.2.1), access to SUT interfaces and associated test actions (Subsection 6.2.2), test component (Subsection 6.2.3), verdict management (Subsection 6.2.4) and test suite (Subsection 6.2.5).

Section 6.3 discusses the manner in which test meta-model concepts were implemented through template-based automatic code generation, by targeting $PL_5$ functionalities. We present automatically generated code and templates examples for our partial implementation of: the `ProviderData` (Subsection 6.3.1), `SystemUnderTest` (Subsection 6.3.2), `TestCase` (Subsection 6.3.3), `TestComponent` (Subsection 6.3.4) and `TestArchitecture` (Subsection 6.3.5) concepts. Verdict management is discussed together with the test case and test component implementation.

Section 6.4 concludes this chapter.

## 6.1 Acceleo Model-to-Text Tool

Acceleo is an implementation of the Object Management Group (OMG) [129] Meta-Object Facility (MOF) [130] Model to Text Language (MTL) [131] standard. In our case, it takes as input: a test model together with the automatic code generation templates. It produces as output: files in the target test language. An automatic code generation template defines the mapping between the concepts from the test meta-model and the test language file structure and code. The template is actually a text with gaps, where for each individual test model the gaps are filled differently. The various templates are organized within Acceleo modules. The architecture of the Acceleo tool, together with the test meta-model, the test models and the templates is shown in Figure 6.1.

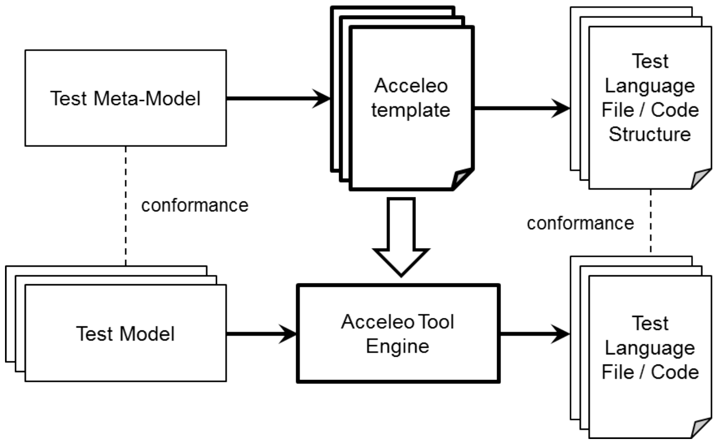


**Figure 6.1: Acceleo Tool Achitecture -**

We present the Acceleo tool on an example. We begin by discussing the two modelling layers that are employed: meta-model level and model level. Next we present the corresponding code and files that should be automatically generated. Finally, we discuss the transformation between the model layer and the code by means of automatic code generation templates with the Acceleo tool.

We consider a simplified meta-model for the definition of Python classes, with two EClass elements (`PythonClass` and `ClassAttribute`) and two EReference elements (`HasClassAttributes` and `InheritanceRelation`). Notice that the `HasClass-Attributes` EReference is a Containment relation. The diagram for this meta-model is shown in Figure 6.2. We chose this example as the target test language for implementation of test models is based on Python.

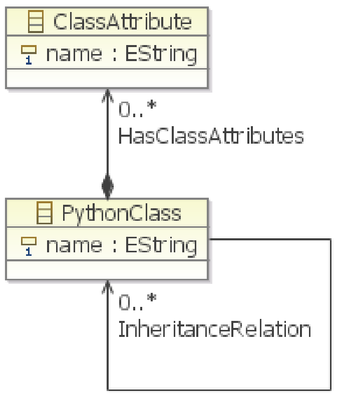


**Figure 6.2: Meta-Model Level: PythonClass/ClassAttribute EClass Elements -**

With this meta-model it is possible to define a model for the following classic example: the *Person* and *Student* classes (Figure 6.3). The Student `PythonClass` of the model has an `Inheritance` with the Person `PythonClass`. The Person `PythonClass` has the `HasClassAttributes` EReference association with the *name* and respectively *surname* `ClassAttribute` elements, while the Student `PythonClass` adds a `HasClass-Attribute` EReference association with the *major* `ClassAttribute` element.

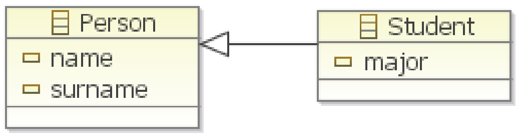


**Figure 6.3: Model Level: Person/Student Python Classes -**

For this model, the desired code is shown in Listing 6.1. Each class is defined in its own Python module.

```

# Person.py Python module
class Person():
    name = ""
    surname = ""

# Student.py Python module
import Person
class Student(Person):
    major = ""
```

**Listing 6.1:** Python Modules and Code

In order to automatically generate this desired code, we define an architecture of three Acceleo modules: one main module (main.mtl) and two modules corresponding to the `PythonClass` and `ClassAttribute` EClass elements (respectively PythonClass.mtl and ClassAttribute.mtl in Figure 6.4). Each module can contain one or several templates. We chose to regroup templates concerning a specific meta-model element within a same module. In this toy example, each module contains only one template.

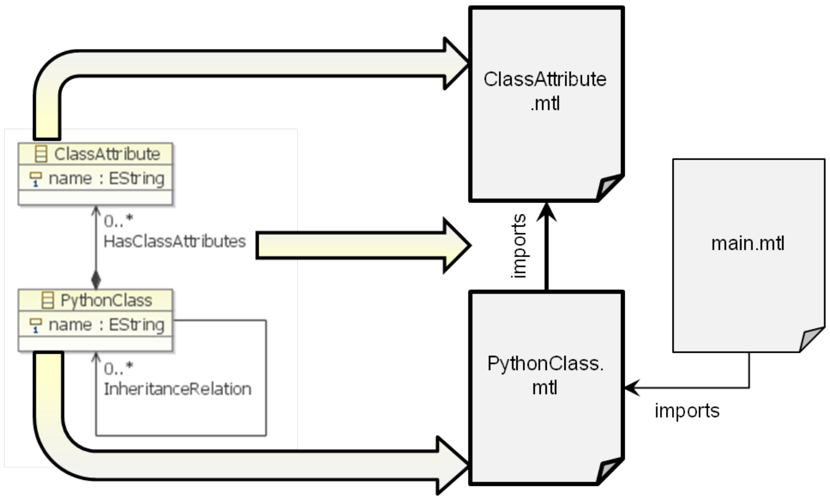


**Figure 6.4: Link between Example Meta-Model Classes and Modules/Templates -**

Let us look at the main module (main.mtl in Listing 6.2). The generateDatabase template (Lines 8-15) has a simple behaviour: for each `PythonClass` element inside the model it calls the PythonClass template. Notice the manner in which the model is explored inside the Acceleo templates: Acceleo statements are given inside the bracket-delimited structures: "[ ... /]". For example, the model is browsed with the [for ...] ... [/for] construct. In the case of our example model, the PythonClass template is called twice: once for the Person and once again for the Student. Notice that the main module needs to import the PythonClass module before calling the template it contains.

Information from the model is extracted and then inserted into the automatically generated code. The text outside the bracket-delimited structures is not interpreted by the Acceleo tool; it is inserted “as is”into the automatically generated code.

```
[comment] main.mtl Acceleo module [/comment]

[comment encoding = UTF-8 /]
[module main('PythonClassExemple')]

[import PythonClass /]

[template public generateDatabase(aDatabase : Database)]
[comment @main/]

[for (aPythonClass : PythonClass | aDatabase.HasPythonClass)]
[aPythonClass.generatePythonClass()/]
[/for]

[/template]
```

**Listing 6.2:** main.mtl

For each `PythonClass`, the generatePythonClass template inside the PythonClass.mtl module (Listing 6.3) generates the corresponding Python module (Person.py and Student.py) and a class skeleton. The generation of Python module files is indicated by the following structure: [file ... ]. In case the`PythonClass` for which the Python module is generated has an `InheritanceRelation` with another `PythonClass`, the necessary import statements are automatically generated as well. This is the case for the Student `PythonClass`. The $i$ variable (Line 17) is implicit and corresponds to the current loop index inside the repetition statement. For clarity issues we did not reproduce in these Listings the exact page setting of the different templates. Notice that this issue of page setting is extremely important, as Python is a language for which the page setting in meaningful: scoping is defined by means of indentations in the code.

In case the `PythonClass` for which the Python module is generated has one or more `ClassAttribute` elements, for each one of these elements the ClassAttribute template is called (Listing 6.4). For the Person `PythonClass` the ClassAttribute template is called twice, while for the Student `PythonClass` it is called only once. Notice that the ClassAttribute template does not generate any Python modules, as it serves only to fill in the Python modules that were already generated by the PythonClass template.

```
[comment] PythonClass.mtl Acceleo module [/comment]

[comment encoding = UTF-8 /]
[module PythonClass('PythonClassExemple')]
[import ClassAttribute /]

[template public generatePythonClass(aPythonClass : PythonClass)]
[file (aPythonClass.name.concat('py'), false, 'UTF-8')]

[for (inheritedPythonClass : PythonClass | aPythonClass.
    InheritanceRelation)]
import [inheritedPythonClass.name/]
[/for]

class [aPythonClass.name/]
[for (inheritedPythonClass : PythonClass | aPythonClass.
    InheritanceRelation)]
[inheritedPythonClass.name/]
[if (i < aPythonClass.InheritanceRelation->size())],[/if]
[/for])
[for (aClassAttribute : ClassAttribute | aPythonClass.HasClassAttributes)]
[aClassAttribute.generateClassAttribute()/]
[/for]

[/file]
[/template]
```

**Listing 6.3:** PythonClass.mtl

```
[comment] ClassAttribute.mtl Aceleo module [/comment]

[module ClassAttribute('PythonClassExemple')]

[template public generateClassAttribute(aClassAttribute : ClassAttribute)]

[aClassAttribute.name/] = ""

[/template]
```

**Listing 6.4:** ClassAttribute.mtl

The approach that we presented begins with the definition of a model and of the corresponding code and files. Only afterwards do we define the automatic code generation modules/templates that produce the code and files from the model. This is a well known approach in literature that has shown its benefits [132]. We confirm that first defining the targeted code and files eases the implementation of the model.

EReference elements with a true value for the Containment (e.g., `HasClassAttributes` between `PythonClass` and `ClassAttribute`) appear at Acceleo module level as import statements (see Figure 6.4 and Listing 6.3). EReference elements with a false value for the Containment (e.g., `InheritanceRelation` between two `PythonClass` elements) appear as import statements as well, but at Python module level (see Listing 6.1). The projection of the meta-model architecture on Acceleo modules/templates and automatically generated code files can aid in the maintenance of the automatic code generation functionality. Engineers can easily find an existing Acceleo module that implements a specific meta-model element. When adding new Acceleo templates, they are to be added by respecting this constraint. The links between automatically generated code files can be more easily identified as well. These same organization principles were employed for our demonstrator, but at a larger scale.

The fact that the test meta-model integrates concepts that are present in existing test languages decreases the difference in level of abstraction between the test meta-model and the test language targeted for implementation. Work such as [133] resolves issues like this one by separating the automatic generation of code and its packaging within files. In our case, we did not consider such an approach to be necessary because of the closeness between the concepts of the test meta-model and those of the target test language. The mapping of the test meta-model architecture at automatic code generation module/template level and Python module level sufficed as an organization principle.

In this section, we have discussed the Acceleo tool and the different functionalities it offers for automatic code generation. The concepts presented herein were used for the implementation of the test meta-model, but at a larger scale than for the toy example that was discussed.

## 6.2 Target Python-Based Test Language ($PL_5$)

We discuss in this section a subset of the functionalities and concepts offered by the target test language $PL_5$, which is based on the scripting programming language Python:

- Test Case (Subsection 6.2.1),
- Access to SUT Interfaces and Associated Test Actions (Subsection 6.2.2),
- Test Component (Subsection 6.2.3),
- Verdict Management (Subsection 6.2.4),
- Test Suite (Subsection 6.2.5).

Although mainly focusing on these $PL_5$ functionalities, the discussion also gives some hints on which test meta-model concepts were mapped onto which $PL_5$ functionalities. This mapping issue is elaborated further on in Section 6.3.

We chose to target a test language outside of the initial sample of test languages ($PL_1$ to $PL_4$) in order to see how our test meta-model would accommodate a new test language. At the time of the test language sample analysis (Chapter 4), $PL_5$ did not exist. Moreover, $PL_5$ lacks some of the functionalities offered by other test languages. This allowed us to analyse how functionalities not offered or only partially offered by a test language can be handled.

In case $PL_5$ does not natively offer a functionality present in the test meta-model, offers it only partially or in a manner inconsistent with the test meta-model, we propose the following solutions:

- implement the functionality or the missing parts using low-level functionalities (e.g., test component implemented with Python multi-threading capabilities),
- generate a wrapper around existing functionalities (e.g., $PL_5$ instructions that access SUT interfaces using their string identifier are wrapped inside navigable strongly-typed object-oriented structures) and code that employs the wrapper,
- define model-to-text generation only for test models that are compatible with the offered functionality (e.g., $PL_5$ offers test monitors whose only possible action is to change the test verdict),
- abandon altogether the implementation of the functionality (e.g., access to SUT low-level interfaces such as physical buses is not offered by $PL_5$).

The second solution was interesting in order to minimize the differences between the test meta-model and $PL_5$, for an easier definition and maintenance of the automatic code generation templates.

We present the various functionalities offered by $PL_5$, by discussing also the manner in which they were linked with test meta-model concepts.

### 6.2.1 Test Case

A test case in $PL_5$ is a Python class. It is a specialization of the predefined *ATLTestCase* class. Listing 6.5 presents the Python module defined in $PL_5$ corresponding to a test case skeleton. The name of the Python module and of the Python class for the test case is the same. These conditions must be met in order to ensure that the $PL_5$ execution engine is capable of comprehending and executing the test case. Test engineers define the desired behaviour inside the three existing methods (`initialize()`, `execute()` and `finalize()`). They are executed sequentially by the external $PL_5$ execution test engine.

We could directly map the `TestCase` concept from the test meta-model to the ATLTestCase class specializations in $PL_5$.

```
# aTestCase.py Python module

from utest.ATL import *

class aTestCase(ATLTestCase):
    def initialize(self):
        # Initialization section
        pass

    def execute(self):
        # Execution section
        pass

    def finalize(self):
        # Finalization section
        pass
```

**Listing 6.5:** aTestCase.py

### 6.2.2 Access to SUT Interfaces and Associated Test Actions

We discuss in this subsection the manner in which access to ICD elements is achieved in PL$_5$, as well as the associated test actions that can be called. We consider application parameters only, as PL$_5$ does not offer access to lower ICD levels. We mapped the `TestAction` concept in the test meta-model to the various instructions offered by PL$_5$.

#### 6.2.2.1 Simple Test Actions

For simple interaction with the SUT, PL$_5$ offers access to the dictionary of application parameters (similarly to PL$_3$). This dictionary is called VsUtils.variables. The application parameters are distinguished by means of their string identifiers. Listing 6.6 shows hot to get and set a parameter value.

While the test meta-model offers a clear object-oriented organization of the SUT interfaces, this is not the case with PL$_5$. In order to bridge this gap and facilitate the definition and maintenance of Acceleo templates, we developed wrappers around the SUT access instructions offered by PL$_5$. These wrappers reconstruct in PL$_5$ the organization of the test meta-model: navigable strongly-typed tree-like SUT interface objects and test actions as associated methods. Listing 6.7 shows a simple wrapper.

```
# Get value
oldValue = VsUtils.variables['SUT_1/ARINC_429_1/LABEL_1/AC_SPEED']

#Set value
VsUtils.variables['SUT_1/ARINC_429_1/LABEL_1/AC_SPEED'] = newValue
```

**Listing 6.6:** Simple Test Actions in PL$_5$

```
# The wrapper
SUT_1.ARINC_429_1.LABEL_1.AC_SPEED.setValue(newValue)

# The wrapped target test language instruction called by setValue()
VsUtils.variables['SUT_1/ARINC_429_1/LABEL_1/AC_SPEED'] = newValue
```

**Listing 6.7:** Wrapper Example in $PL_5$

#### 6.2.2.2 Time-Dependent Test Actions

For time-dependent interaction with the SUT, $PL_5$ offers predefined instructions (e.g., for sine, ramp or square signals). These interactions do not block the execution of the caller thread, being executed in parallel. The application parameters are distinguished by means of their string identifiers. An example is given in Listing 6.8.

```
# Time-dependant test action declaration on SUT interface
signalSinus = ATLInjectionVSSinus('SUT_1/ARINC_429_1/LABEL_1/AC_SPEED',
    amplitude=10, period_ms=10000, phase_rad=math.pi*0.5, sampling_ms=50)

# Time-dependant test action execution control
signalSinus.start()
time.sleep(duration)
signalSinus.stop()
```

**Listing 6.8:** Time-Dependant Test Actions in $PL_5$

In the above example the desired signal is first defined, together with the SUT interface on which it is to be applied. Afterwards, the `start()` command begins the generation of the signal in a separate execution thread. The generation of the signal is halted by the `stop()` command.

Notice that the duration of the time-dependent test action does not appear within the list of parameters: a `time.sleep()` instruction is required. As such, we had to render it explicit within our automatic code generation, so that it corresponds to our classification of `TestAction` elements: `Timed` versus `UnTimed` (Figure 5.2).

The wrappers used for this type of instructions follow the same principles as the ones previously discussed, although they are more complex.

#### 6.2.2.3 Fault Injection Test Actions

For fault injection on SUT application parameters, $PL_5$ employs the instructions such as those shown in Listing 6.9.

```
VsUtils.variables.addVariables(['SUT_1/ARINC_429_1/LABEL_1/AC_SPEED', '
    SUT_1/ARINC_429_1/LABEL_1/AC_ALTITUDE'])

VsUtils.vc.configureInjection('SUT_1/ARINC_429_1/LABEL_1/AC_SPEED', "
    FixedValue", injectionParams)
```

**Listing 6.9:** Fault-Injection Test Actions in $PL_5$

First, the application parameters on which fault injection is to be applied must be added to a specific set with the `addVariables` command. Afterwards, the fault injection is configured for each application parameter in this set with the `configure-Injection` command. In the above example, the fault injection consists in forcing a value for an application parameter.

As previously discussed, this type of instructions were also wrapped and attached to SUT interface objects.

### 6.2.3 Test Component

$PL_5$ does not offer the concept of test component, with the exception of a simplified test monitor. We enriched $PL_5$ with the `TestComponent` concept of the test meta-model, by employing the native Python multi-threading functionalities in the automatic code generation. We did this for the `SequentialTestComponent`, `PeriodicTestComponent` and `CycleByCycleTestComponent` specializations. We do not discuss it here how these concepts were implemented, for more information refer to Subsection 6.3.4. For the sake of variety, we mapped the `TestMonitor` concept of the test meta-model to the simplified test monitor offered by $PL_5$. We could have implemented the complete test monitor concept using low-level functionalities offered by Python, but decided to try a case with partial implementation of a concept. Only simple test monitors in test models, compatible with the $PL_5$ test monitor, were targeted in our prototype.

In $PL_5$, a test monitor is a class specializing the predefined ATLAsyncTest class (Listing 6.10). It has two simple types of conditions on the value of an application parameter:

- a condition on the last value of an application parameter, at the end of the execution of the test monitor,
- a condition on the values of an application parameter, during the life of the test monitor.

The condition is passed as a string parameter to the test monitor. The resulting behaviour of the test monitor following the activation of the logical condition is implicit: a local verdict is set depending on the evaluation of the condition during/at the end of the life of the test monitor. Test engineers cannot modify this predefined behaviour. In the example in Listing 6.10, if the application parameter has a value higher than an admissible threshold during the life of the test monitor, the local test verdict is automatically set to `Fail`. The activation duration for the test monitor is passed as a parameter to the execute command.

As $PL_5$ does not natively offer test components, it also lacks the concept of test architecture, which we implemented using Python low-level functionalities.

```
asyncExec = ATLAsyncTest(self)

asyncExec.failIfInterval('SUT_1/ARINC_429_1/LABEL_1/AC_SPEED > 100', '
    Message')

asyncExec.execute(5)
```

**Listing 6.10:** Test Monitor in $PL_5$

### 6.2.4 Verdict Management

We have already seen some verdict management functionalities when discussing the test monitor concept in $PL_5$. Verdict management is also possible inside an ATLTestCase. Two types of test verdicts exist in $PL_5$: pass and fail. See the example in Listing 6.11 where the value of an application parameter is verified against an expected value.

Our implementation considers the following test verdicts defined in the test meta-model: `None`, `Pass`, `Fail` and `Error`. The `Pass` and `Fail` values were mapped directly to the ones already offered by $PL_5$. The `Error` verdict was also mapped to the $PL_5$ fail verdict, but we accompanied it with specific error messages so as to differentiate the two. The management of all test verdicts, with the exception of `Inconclusive`, was implemented from scratch (see Sections 6.3.3 and 6.3.4).

```
self.failIf(VsUtils.variables['SUT_1/ARINC_429_1/LABEL_1/AC_SPEED'] ==
    100, 'Message')
```

**Listing 6.11:** Verdict Management in $PL_5$

### 6.2.5 Test Suite

A $PL_5$ test suite is defined outside of the test language itself, inside a GUI that stores it in external configuration files. A $PL_5$ test group is a sequence of test cases (not to be confused with the test meta-model `TestGroup` concept). A $PL_5$ test suite is a file that contains a sequence of test groups and individual test cases. An example of such an external configuration file is given in Listing 6.12. It shows two test groups (containing test cases), as well as two independent test cases.

The `TestSuite` concept in the test meta-model allows the definition of richer execution orders for `TestCase` elements than $PL_5$. Full implementation of the concept would have had to be developed from scratch. As inter-test organization was not one of our major concerns, we chose in our prototype to address only the implementation of `TestSuite` elements compatible with the functionalities offered by the $PL_5$ framework.

The next chapter discusses the manner in which a subset of test meta-model concepts was mapped to functionalities offered by $PL_5$.

```
TestSuite.tsdef test suite configuration file

<TestCases>
  <TestGroup ID="MyTestGroup_1" nbOfExecutions="1" isEnabled="True">
    <TestCase ID="MyTestCase_1" nbOfExecutions="2" isEnabled="True" />
    <TestCase ID="MyTestCase_2" nbOfExecutions="3" isEnabled="True" />
  < /TestGroup>
  <TestGroup ID="MyTestGroup_2" nbOfExecutions="4" isEnabled="True">
    <TestCase ID="MyTestCase_3" nbOfExecutions="5" isEnabled="True" />
  </TestGroup>
  <TestCase ID="MyTestCase_4" nbOfExecutions="6" isEnabled="True" />
  <TestCase ID="MyTestCase_5" nbOfExecutions="7" isEnabled="True" />
</TestCases>
```

**Listing 6.12:** Test Suite in $PL_5$

## 6.3 Architecture of Acceleo Modules/Templates

Before discussing the automatic code generation templates in more detail, let us review the list of test meta-model concepts that were implemented in the demonstrator, together with the target test language $PL_5$ functionalities they were mapped to:

- `ProviderData`: `ConnectionPointType` and associated `TestAction` elements in test meta-model $\rightarrow$ wrapper around $PL_5$ instructions,
- `SystemUnderTest` in test meta-model $\rightarrow$ usage of `ProviderData` wrapper,
- `TestCase` in test meta-model $\rightarrow$ ATLTestCase class specializations in $PL_5$,
- `TestComponent` (except `TestMonitor`) in test meta-model $\rightarrow$ code generation using low-level Python functionalities,
- `TestMonitor` in test meta-model $\rightarrow$ partial implementation towards ATLAsyncTest class specialization in $PL_5$,
- `TestArchitecture` in test meta-model $\rightarrow$ code generation using low-level Python functionalities,
- `TestSuite` in test meta-model $\rightarrow$ partial implementation towards $PL_5$ framework test suites,
- Verdict Management $\rightarrow$ $PL_5$ functionality upgraded by code generation using low-level Python functionalities.

In the demonstrator we only targeted the application parameter level of the ICD. As $PL_5$ does not offer access to buses and messages, it would not have been realistic to develop this functionality from scratch for our prototype.

This section presents some of the Acceleo automatic code generation modules/templates that implement the test meta-model concepts. In Subsection 6.3.1, we cover the wrapper created around $PL_5$ for the `ProviderData` concept. Next we discuss in Subsection 6.3.2 the manner in which the interfaces of the SUT are defined from the "instantiation"of `ProviderData` elements inside `SystemUnderTest` elements. Finally we present the implementation choices for the test case (Subsection 6.3.3), test component (Subsection 6.3.4) and test architecture (Subsection 6.3.5) concepts. We focus on these structural elements as the behavioural elements had a straight forward implementation.

### 6.3.1 ProviderData Implementation

A wrapper is defined for the implementation of the `ProviderData` concept, including SUT interface types, toolkits and attached test actions. The wrapper hides the instructions of the target test language by reproducing the organization of the `ProviderData` using Python classes and methods. This was one of the first architectural choices for the automatic code generation. It allowed us to:

- reduce the abstraction level between the test models and the automatically generated code and consequently simplifying the Acceleo modules/templates,
- facilitate the link between the test model and automatically generated code, for debugging purposes.

We will see next how this is achieved. Listing 6.13 shows the file My_ProviderData.py, created for a hypothetical My_ProviderData `ProviderData` instance in a test model. Let us consider that My_ProviderData contains:

- a FloatApplicationParameter instance of the `ApplicationParameterType` concept, with:
  - the setValue and generateRampSignal instances of the `TestAction` concept,
- a TimeManagementToolkit instance of the `Toolkit` concept, with:
  - the waitDuration instance of the `TestAction` concept.

All test actions have input/output parameters that we do not detail here.

A Python class is defined for each concept in the test meta-model. For example, the *ApplicationParameterType* Python class is created for the `ApplicationParameter-Type` EClass (Listing 6.13, Line 7). Afterwards, for each instance of the said concept inside the test model, specialization classes are defined. For example, for the *FloatApplicationParameter* instance of the `ApplicationParameterType` EClass, a Python class with the same name is created specializing the ApplicationParameterType Python class (Line 15).

Observe that the instantiation relation between the `ApplicationParameterType` in the test meta-model and the `FloatApplicationParameter` in the test model is translated into a specialization relation between the `ApplicationParameterType` Python class and the `FloatApplicationParameter` Python class. This is linked to our implementation of the Type Object design pattern [127]: classes are created for the type (in the test meta-model) and the type instances (in the test model), with specialization relations between them. Thus, some instantiation relations between the test meta-model and test models are transformed into specialization relations at code level.

A number of methods are attached to the Python classes, corresponding to the associated test actions. As previously discussed, some test actions are attached to SUT interface types (Lines 19-22), while others are attached to tookits (Listing 6.13, Line 25). These methods hide the implementation details in $PL_5$, making it resemble the test meta-model. For example, the *generateRampSignal* method (corresponding to the test action in the test model with the same name) hides the *ATLInjectionVSRamp()* time-dependent test action from $PL_5$ (Subsection 6.2.2.2), acting as a wrapper.

A snippet of the ProviderData.mtl template, which led to the creation of the My_ProviderData.py file, is shown in Listing 6.14. Notice towards the end of the file that additional templates are called: *generateConnectionPointType* and *generateToolkit* (Lines 26-30). The modules containing them were imported at the beginning of the file: *ConnectionPointType* and *Toolkit* (Lines 5, 6). For each concept in the test meta-model we created its own module. Consequently, the architecture of Acceleo modules maps the organization of the test meta-model, similarly to the manner in which this organization is mapped onto the automatically generated files and code. This was another architectural choice for our implementation, which we believe can aid test solution provider engineers to work with the modules and with the automatically generated files and code based on their knowledge of the structure of the test meta-model. Some information inside the template is static (identical to the generated code) (Lines 11-23), while some information is dynamically generated (Lines 25-31).

```
from utest.ATL import *
import time
from threading import Thread

# [From Test Meta-Model] ConnectionPointType EClass Specializations (
    showing here only a subset)
...
class ApplicationParameterType:
  name = ""

class Toolkit:
  name = ""

# [From Test Model] ConnectionPointType First "Instantiation" (showing
    here only a subset)
...
class FloatApplicationParameter(ApplicationParameterType):
  name = "Float Application Parameter"
  instanceName = ""
  fullPath = ""
  def setValue(self,newValue):
    print "(",time.strftime('%H:%M:%S',time.localtime()),")","Executing
        TestAction","setValue()"
    VsUtils.variables[self.fullPath] = newValue
  def generateRampSignal(self,startValue,endValue,DurationValue,
      DurationUnit):
    ...

class TimeManagementToolkit(Toolkit):
  name =  "TimeManagementToolkit"
  def waitDuration(self,DurationValue,DurationUnit):
    ...
```

**Listing 6.13:** My_ProviderData.py Snippet

```
[comment] ProviderData.mtl Aceleo module [/comment]

[module ProviderData('http://eads.ts.ate.hmi.stelae.model/model/stelae.
    ecore')]
[import content::getNameQuery /]
[import ConnectionPointType /]
[import Toolkit /]
[template public generateProviderData(aProviderData : ProviderData)]

[file (aProviderData.name.getName().concat('.py'), false, 'UTF-8')]

from utest.ATL import *
import time
from threading import Thread

# [From Test Meta-Model] ConnectionPointType EClass Specializations (
    showing here only a subset)
...
class ApplicationParameterType:
  name = ""

class Toolkit:
  name = ""

# [From Test Model] ConnectionPointType First "Instantiation" (showing
    here only a subset)
...
[for (aConnectionPointType : ConnectionPointType | aProviderData.
    HasConnectionPointTypes->select(any : ConnectionPointType | any.
    oclIsTypeOf(ApplicationParameterType)))]
[aConnectionPointType.generateConnectionPointType()/]
[/for]

[for (aToolkit : Toolkit | aProviderData.HasToolkits)]
[aToolkit.generateToolkit()/]
[/for]

[/file]
[/template]
```

**Listing 6.14:** ProviderData.mtl Snippet

### 6.3.2 SystemUnderTest Implementation

Here we discuss how ICD element types (e.g., FloatApplicationParamater Python class) presented in the previous Subsection are instantiated in order to define the interfaces of an SUT. Let us consider a My_TestContext instance of the `TestContext` concept, containing a My_SUT instance of the `SystemUnderTest` concept. MySUT contains:

- an ARINC_429_1 `PhysicalBusInstance` of the ARINC_429 `PhysicalBusType`,

- with an ARINC_429_Label_1 `MessageInstance` of the ARINC_429_Label `MessageType`,

- that transports a App_Parameter_1 `ApplicationParameterInstance` of the FloatApplicationParameter `ApplicationParameterType`.

The needed files and code for the various types of SUT interface are automatically generated through the means presented in the previous Subsection (see for example the Python class created for the float type of application parameter in Listing 6.13).

The simplified automatically generated file and code for My_SUT is shown in Listing 6.15. Observe the cascading instantiation: the My_SUT creates an instance of an ARINC_429 physical bus (Line 31), which in turn creates an object for an ARINC_429_Label_(Line 20), and so forth until reaching the application parameter level. This example corresponds to the second instantiation step discussed in Section 5.8. The only element that is not instantiated here is My_SUT. As will be seen in Subsection 6.3.4, it is instantiated by the test components that interact with the SUT. In Listing 6.15, My_ProviderData is imported at the beginning so as to have access to the Python classes defined for the different SUT interface types (Line 1). The fullPath attribute (Listing 6.15 - Lines 7, 18, 30) is used to store the string identifier for each ICD element.

The simplified Acceleo module/template that created the My_SUT.py file can be found in Listing 6.16. The cascade instantiation by the My_SUT for all the physical buses it contains can be observed in Lines 22-24, while the instantiation by the ARINC_429_1 for all the messages in contains can be observed in Lines 12-14. Notice also the creation of the fullPath attribute by concatenating several SUT interface element names (Lines 9, 20), while exploring the test model in a direction inverse to the containment direction (a `SystemUnderTest` owns a `PhysicalBusInstance`, while this `PhysicalBusInstance` knows who its owner is through the `IsOwnedBySystemUnderTest` EReference).

```
import My_ProviderData

# The ApplicationParameter Elements

class Instance_App_Parameter_1(MyProviderData.FloatApplicationParameter):
  instanceName = "My_App_Parameter_1"
  fullPath = "My_SUT/ARINC_429_1/ARINC_429_Label_1/App_Parameter_1"
  permanentIdentifier = ""

# The MessageInstance Elements
...
# The LogicalBusInstance Elements
...
# The PhysicalBusInstance Elements

class Instance_ARINC_429_1(MyProviderData.ARINC_429):
  instanceName = "ARINC_429_1"
  fullPath = "My_SUT/ARINC_429_1"
  permanentIdentifier = ""
  ARINC_429_Label_1 = Instance_ARINC_429_Label_1()

# The PinInstance Elements
...
# The ConnectorInstance Elements
...
# The SystemUnderTest Element

class Instance_My_SUT(MyProviderData.CPIOM):
  instanceName = "My_SUT"
  fullPath = "My_SUT"
  ARINC_429_1 = Instance_ARINC_429_1()
```

**Listing 6.15:** My_SUT.py Snippet

```
...
import [aSystemUnderTest.IsOfSystemUnderTestType.IsOwnedByProviderData.
    name.getName()/]
...
# The PhysicalBusInstance Elements

[for (aPhysicalBusInstance : PhysicalBusInstance | aSystemUnderTest.
    HasPhysicalBusInstances)]
class Instance_[aPhysicalBusInstance.name.getName()/]([
    aPhysicalBusInstance.IsOfPhysicalBusType.IsOwnedByProviderData.name.
    getName()/].[aPhysicalBusInstance.IsOfPhysicalBusType.name.getName()
    /]):
  instanceName = "[aPhysicalBusInstance.name/]"
  fullPath = "[aPhysicalBusInstance.IsOwnedBySystemUnderTest.name.getName
      ()/]/[aPhysicalBusInstance.name.getName()/]"
  permanentIdentifier = "[aPhysicalBusInstance.PermanentIdentifier/]"
  ...
  [for (aMessageInstance : MessageInstance | aPhysicalBusInstance.
      HasMessageInstances)]
  [aMessageInstance.name.getName()/] = Instance_[aMessageInstance.name.
      getName()/]()
  [/for]
[/for]
...
# The SystemUnderTest Element
class Instance_[aSystemUnderTest.name.getName()/]([aSystemUnderTest.
    IsOfSystemUnderTestType.IsOwnedByProviderData.name.getName()/].[
    aSystemUnderTest.IsOfSystemUnderTestType.name.getName()/]):
  instanceName = "[aSystemUnderTest.name/]"
  fullPath = "[aSystemUnderTest.name.getName()/]"
  ...
  [for (aPhysicalBusInstance : PhysicalBusInstance | aSystemUnderTest.
      HasPhysicalBusInstances)]
  [aPhysicalBusInstance.name.getName()/] = Instance_[aPhysicalBusInstance.
      name.getName()/]()
  [/for]
[/file]
[/template]
```

**Listing 6.16:** SystemUnderTest.mtl Snippet

### 6.3.3 TestCase Implementation

We discuss here the implementation of the test case concept of the test meta-model.

Let us consider that the My_TestContext discussed in the previous Subsection also contains a My_TestCase and a My_SequentialTestComponent. The latter is instantiated once inside the test case, the name of the instance being My_SequentialTestComponent_1. This instance is started within the My_TestSection of the test case. The corresponding automatically generated file and code is shown in Listing 6.17.

Notice the manner in which we reuse the ATLTestCase class from $PL_5$ (previously discussed in Subsection 5.4.2):

- My_TestCase is a specialization of the the ATLTestCase class (Line 7),
- we implement the predefined ATLTestCase class methods: initialize(), execute() and finalize() (Lines 16, 20, 35).

First an instance of My_SequentialTestComponent is defined (Line 10). Within the execute() method, inside the My_TestSection indicated by a comment (Line 24), the test component instance is started (Line 25). The end of this method presents part of our implementation of the global test verdict synthesis from local test verdicts: the global test verdict of the test case is synthesized from the local test verdicts of the test component instances (Listing 6.17, Lines 27-31). In this simplified example, the global test verdict of the test case is equivalent to the local test verdict of its sole test component instance. Notice the import statements at the beginning (Listing 6.17, Lines 2-5). The test context is necessary as the test component instance needs to be made aware of the environment in which it is executed (Line 11), so as to have access to the SUT interfaces, shared data pool and events. The provider data is necessary so that the test case be capable of calling test actions on toolkits, although our simplified example does not take advantage of this functionality. The names of the Python modules to import are derived by exploring the relations of the test meta-model, in the same manner in which the fullPath attribute was previously constructed.

We do not discuss here the corresponding Acceleo automatic code generation template, as it does not bring any new information in comparison to those templates we already presented.

```
from utest.ATL import *
import time
import My_TestContext
import My_ProviderData
import My_SequentialTestComponent

class My_TestCase(ATLTestCase):
  TestCaseVerdict = 0
  MyTestContext = MyTestContext.MyTestContext()
  My_SequentialTestComponent_1 = My_SequentialTestComponent.
      My_SequentialTestComponent()
  My_SequentialTestComponent_1.TestContext = MyTestContext
  My_SequentialTestComponent_1.InstanceName = "
      My_SequentialTestComponent_1"

  # TestArchitecture Implementation
  ...
  def initialize(self):
    print "STELAE Online.."
    pass

  def execute(self):
    self.My_SequentialTestComponent_1.TestCase = self
    print "(",time.strftime('%H:%M:%S',time.localtime()),")","Starting
        TestCase ","My_TestCase"

    # My_TestSection
    self.My_SequentialTestComponent_1.StartExecutableElement()
    while(not(self.My_SequentialTestComponent_1.ExecutionStatus=="
        ExecutionStopped")): pass
    self.TestCaseVerdict = self.My_SequentialTestComponent_1.
        TestComponentVerdict
    if (self.TestCaseVerdict == 0):
      # self.failIf(True, "NONE")
      print "(",time.strftime('%H:%M:%S',time.localtime()),")","[NONE]","
          Verdict of TestCase","My_TestCase"
    ...
    print "(",time.strftime('%H:%M:%S',time.localtime()),")","Stopping
        TestCase ","My_TestCase"
    pass

  def finalize(self):
    print "STELAE Offline.."
    pass
```

**Listing 6.17:** My_TestCase.py Snippet

### 6.3.4 TestComponent Implementation

In this Subsection we discuss the implementation of the test component concept. This functionality was not natively offered by $PL_5$ and consequently we had to add it through our code generation, using native Python multi-threading instructions.

Let us consider My_SequentialTestComponent that we previously discussed. We assume that its behaviour corresponds to a simplified auto-test scenario and is separated into two sequential blocks: Initialization and Verification. Initialization sets the value of App_Parameter_1 to zero and afterwards pauses its execution for one second. Verification sets the value of App_Parameter_1 to one and verifies after one second that the initial value has changed. Listing 6.18 presents the corresponding automatically generated file and code. Notice that the Python class corresponding to the test component is a specialization of `threading.Thread` (Line 5), which allows us to launch the execution of each test component in a separate thread. Our test component also inherits the `run()` (Line 24) and `start()` (Line 17) methods from thread.Threading. The `start()` method launches the execution of the `run()` behaviour in a separate thread. We added three new interfaces to the test component, taken from the test meta-model: `Start/Pause/StopExecutableElement()` (Lines 14, 20, 22). `StartExecutableElement()` acts as a wrapper for the `start()` method. We also added an ExecutionStatus attribute for the test component (Line 11), needed for implementing from scratch the manner in which the execution of a test component is controlled. Notice how the test sections in the model are transformed into code annotations (Lines 27, 31).

The end of the `run()` method shows our implementation of the global test verdict synthesis from local test verdicts: the test verdict of the test component is derived from the verdicts of all the checks (Lines 34-39). The catching of exception leads to the `Error` test verdict (Lines 41-44).

The periodic and cycle-by-cycle test components have a more complex execution control that manages the duration of the code execution, while setting an error local verdict in case the code execution duration is not respected by the test platform.

Because of the size of the Acceleo automatic code generation modules/templates for the test component concept, we do not present them here. We tried to reuse as much as possible of the templates for all test component types. All templates for test component are identical, with the exception of the behavioural part that appears inside the run() method, which is specific to each test component type. We do not discuss the test monitor implementation because of the simplicity of reusing the $PL_5$ monitor construct.

As you will see later on (Chapter 7), the test model is much more concise and user-friendly than the generated code.

```python
...
import My_ProviderData
import My_SUT

class My_SequentialTestComponent(threading.Thread):
  My_SUT = My_SUT.Instance_My_SUT()
  TimeManagementToolkit = MyProviderData.TimeManagementToolkit()

  # Accessor Declaration
  ...
  ExecutionStatus = "ExecutionStopped"
  TestComponentVerdict = 0

  def StartExecutableElement(self):
    if (self.ExecutionStatus != "ExecutionRunning"):
      self.ExecutionStatus = "ExecutionRunning"
      self.start()
    else:
      print "TestComponentInstance",self.InstanceName,"Already Running"
  def PauseExecutableElement(self):
    ...
  def StopExecutableElement(self):
    ...
  def run(self):
    print "(",time.strftime('%H:%M:%S',time.localtime()),")","Starting TestComponentInstance", self.InstanceName
    try:
      # Initialization
      self.My_SUT.ARINC_429_1.ARINC_429_Label_1.App_Parameter.setValue(0)
      self.TimeManagementToolkit.waitDuration(1,"second")

      # Verification
      self.My_SUT.ARINC_429_1.ARINC_429_Label_1.App_Parameter.setValue(1)
      self.TimeManagementToolkit.waitDuration(1,"second")
      if (self.My_SUT.ARINC_429_1.ARINC_429_Label_1.App_Parameter.getValue()==1):
        print "(",time.strftime('%H:%M:%S',time.localtime()),")","[PASS] self.My_SUT.ARINC_429_1.ARINC_429_Label_1.App_Parameter.getValue()==1"
        self.TestComponentVerdict = max(1, self.TestComponentVerdict)
        self.TestCase.failIf(False, "PASS")
      else:
        ...
      pass
    except:
      print "(",time.strftime('%H:%M:%S',time.localtime()),")","[ERROR]", self.InstanceName
      traceback.print_exc(file=sys.stdout)
      self.TestComponentVerdict = 4
    print "(",time.strftime('%H:%M:%S',time.localtime()),")","Stopping TestComponentInstance", self.InstanceName
    self.ExecutionStatus = "ExecutionStopped"
```

**Listing 6.18:** My_SequentialTestComponent.py Snippet

### 6.3.5 TestArchitecture Implementation

We discuss in this subsection the implementation of the test architecture concept. Let us consider that My_SequentialTestComponent has an My_Accessor formal interface for float application parameters. All of the behaviour of the test component is defined on this accessor, while the test architecture specifies on which SUT interface this behaviour is reflected. Listing 6.19 shows the definition of the accessor inside the automatically generated code for My_SequentialTestComponent (Line 4). The accessor is a strongly-typed object, having access to all the test actions (i.e., methods) owned by the SUT interface type to which it connects (this is the reason why we implemented the accessor as an instance of connection point type). Listing 6.20 shows the template that was used for generating this accessor definition. Once defined, the accessor is used in the Initialization section (Listing 6.19, Line 9), replacing the previous direct access to the application parameter (Listing 6.18, Line 28).

```
  class My_SequentialTestComponent(threading.Thread):
2   ...
    # Accessor Declaration
4   My_Accessor = My_ProviderData.FloatApplicationParameter()
    ...
6   def run(self):
      ...
8     # Initialization
      self.My_Accessor.setValue(0)
```

**Listing 6.19:** My_SequentialTestComponent.py - ApplicationParameterAccessor Snippet

```
[module EngineerVariableAccessor('http://eads.ts.ate.hmi.stelae.model/
    model/stelae.ecore')]
[import content::getNameQuery /]

[template public generateEngineerVariableAccessor(
    aEngineerVariableAccessor : EngineerVariableAccessor)]
[aEngineerVariableAccessor.name.getName()/] = [aEngineerVariableAccessor.
    ReferencesEngineerVariableType.IsOwnedByProviderData.name.getName()
    /].[aEngineerVariableAccessor.ReferencesEngineerVariableType.name.
    getName()/]()
[/template]
```

**Listing 6.20:** EngineerVariableAccessor.mtl Snippet

We consider that the test architecture specifies a connection between My_Accessor and App_Parameter_1 of the SUT. The accessor is defined in the Python class corresponding to the test component, while the connection is in the Python class corresponding to the test case. Listing 6.21 shows the definition of the connection inside the automatically generated code for My_TestCase (Line 4). This is achieved by giving the accessor the identifier of the SUT interface, through the fullPath attribute. Listing 6.22 shows the template used for assigning the value of the fullPath attribute, by traversing the tree-like ICD structure and concatenating the names of the interfaces, in reverse order of containment. The various [comment] .. [/comment] blocks where used only to spread the template code onto several lines, so as to enhance its clarity.

```
class My_TestCase(ATLTestCase):
  ...
  # TestArchitecture Implementation
  My_SequentialTestComponent_1.My_Accessor.fullPath = "My_SUT/ARINC_429_1/
      ARINC_429_Label_1/App_Parameter_1"
```

**Listing 6.21:** My_TestCase.py - TestArchitecture Snippet

```
[module ConnectionPointAccessorConnection('http://eads.ts.ate.hmi.stelae.
    model/model/stelae.ecore')]
[import content::getNameQuery /]

[template public generateConnectionPointAccessorConnection(
    aConnectionPointAccessorConnection : ConnectionPointAccessorConnection
    )]
[if (aConnectionPointAccessorConnection.
    ProvidesAccessToConnectionPointInstance.oclIsTypeOf(
    EngineerVariableInstance))]
[aConnectionPointAccessorConnection.ForTestComponentInstance.name.getName
    () /].[comment]
[/comment][aConnectionPointAccessorConnection.
    ProvidesAccessForConnectionPointAccessor.name.getName()/].fullPath = [
    comment]
[/comment]"[aConnectionPointAccessorConnection.
    ProvidesAccessToConnectionPointInstance.oclAsType(
    EngineerVariableInstance).IsOwnedByMessageInstance.
    IsOwnedByPhysicalBusInstance.IsOwnedBySystemUnderTest.name.getName()
    /]/[comment]
[/comment][aConnectionPointAccessorConnection.
    ProvidesAccessToConnectionPointInstance.oclAsType(
    EngineerVariableInstance).IsOwnedByMessageInstance.
    IsOwnedByPhysicalBusInstance.name.getName()/]/[comment]
[/comment][aConnectionPointAccessorConnection.
    ProvidesAccessToConnectionPointInstance.oclAsType(
    EngineerVariableInstance).IsOwnedByMessageInstance.name.getName()/]/[
    comment]
[/comment][aConnectionPointAccessorConnection.
    ProvidesAccessToConnectionPointInstance.oclAsType(
    EngineerVariableInstance).name.getName()/]"
[/if]
[/template]
```

**Listing 6.22:** ConnectionPointAccessorConnection.mtl Snippet

## 6.4 Conclusion

In this Chapter we presented the implementation of some of the concepts of the test meta-model (test case, test component, SUT interfaces and interactions, test verdict management), through model-to-text template-based automatic code generation. We targeted a new test language that we called $PL_5$, which is executable on the U-TEST Real-Time System [3] integration test platform. The Acceleo tool [125] was used for the automatic code generation. For our implementation, we took the structure of the test meta-model as an organizing principle of the Acceleo modules/templates, as well as of the automatically generated Python files and code. We found that this approach was helpful and allowed us to better navigate the rich set of domain-specific concepts. We also showed how information from test models is handled by the Acceleo modules/templates in order to implement the chosen concepts. We targeted functionalities already offered by $PL_5$ (e.g., test case), functionalities that were lacking (e.g., test component), as well as functionalities that were partially offered (e.g., test monitor).

The automatic code generation from test models to test language files and code is quasi-instantaneous.

A well-known approach for an easy and rapid definition of automatic code generation templates is to first select a source simple example (we chose a test case from a case study that we will discuss in Chapter 7), then define the expected target (what the generated files and code would be) and only afterwards develop the templates that map the two [132]. Our experience confirms this, as we encountered no difficulty when developing the templates while being guided by the use case. Currently, around 40% of the concepts present in the test meta-model have been implemented. The missing concepts were not implemented as the simple/medium complexity case study did not require them. Moreover, we targeted a test language that only offered access to the application parameter level of the SUT interfaces. Consequently, concepts related to the bus and message ICD hierarchical levels could not be implemented. For the implemented concepts we defined a total of seventy-five Acceleo modules, each with one template. A total effort of one person-month was required for the definition of the modules/templates.

The implementation did not raise major questions concerning the test meta-model, which did not suffer many modifications. The only problems that were identified concerned information inside the test meta-model which was not always directly accessible from a given concept. For example, as test components can call toolkits, they need to have access to the test solution provider section where the toolkits are declared. In the test meta-model no direct relation exists between the two. A path can nevertheless be found linking the two, by traversing the relations between the test component, the test case that instantiates it, the test engineer section and finally the test solution provider section. Some of the relations on this path were initially uni-directional in the test meta-model, and consequently for a given test solution provider section we could find all the test components linked to it, but not the other way around. The implementation helped us identify such situations and complete the test meta-model by rendering some relations bi-directional. The implementation did not raise major questions concerning the test languages analysis either. As $PL_5$ did not exist when we finalized our test languages analysis, its usage for the implementation of our approach allowed us to challenge the genericity of our analysis, as well as of our test meta-model. Had $PL_5$ existed at that time, it would not have changed the outcomes of our test languages analysis, nor the structure of the test meta-model.

The approach based on model-to-text transformations and structured templates sufficed for our implementation. We did not consider it useful to use more complex techniques for implementing the test models, such as those discussed in [132] or [133] (e.g., a two-step transformation: generation of unstructured code followed by its organization within files, useful when one part of the model acts as a source to several pieces of code). In addition, model-to-model transformations were not considered, as none of the test languages to which we had access had a corresponding meta-model that we could target. Our approach was motivated by the similarity between the test meta-model concepts and the target test language functionalities. Moreover, our organization of the different domain-specific concepts is inspired from what the test languages offer and as such sometimes direct mappings can be defined from test models to test language files/code. In any case, we believe that the following minimal functionalities need to be offered by a target programming language in order to implement our test models without much difficulty:

- access to the SUT interfaces by employing ICD-derived identifiers,
- an object-oriented programming paradigm,
- multi-threading capabilities.

We would also like to mention an issue that is important in an industrial environment: debugging. This topic can be discussed from two complementary points of view. The first concerns the debugging of the automatic code generation templates by test solution provider engineers. The second concerns the debugging of test models by test engineers (i.e., test engineers do not debug the automatically generated code, in conformance with the model-driven engineering philosophy).

As regards faults inside automatic code generation templates, we currently do not have a validation solution. This would be a difficult issue to resolve due to the fact that the target proprietary test languages do not have a formal semantics or implicitly verified properties (as in the case of the SCADE Suite [134]). We tried to alleviate the problem by proposing:

- wrappers around target test language instructions, based on the textual representation of the test meta-model, in order to enhance the clarity of the link between the automatically generated code and the textual representation (going even further with the wrapper method by defining additional ones would be possible),
- an agile use-case-based approach, where automatic code generation templates are defined and verified based on a use-case containing the input model and the expected output files/code,
- a test meta-model-based architecture of automatic code generation templates and automatically generated files/code, that eases their definition, navigation and maintenance (in addition, the Acceleo environment offers auto-completion facilities for navigating the meta-model and inserting the required data inside the templates).

With regard to faults inside test models, this issue is common to all model-driven approaches, as existing debugging tools were developed to work at code level. For our specific approach, various solutions have been identified:

- tackle the debug issue at test model level, for example by inserting the breakpoint concept inside the test meta-model, employing annotation-based round-trip engineering techniques, adding extra OCL rules or developing a specific tool,
- bring closer together the grammars of the textual representation of the test meta-model behavioural part and of the target test language: by modifying the concrete test model syntax and adding wrappers (we did this but there were limitations: the concrete syntax we were able to define was not exactly that of Python),
- a more radical solution (allowed by the modularity of the test meta-model) would be to decouple the structural and behavioural aspects of the test meta-model, with the templates previously presented being reused for the structural part, while the behavioural part being defined directly in the target test language. This solution would forfeit the fault avoidance, customization and portability features the test meta-model currently offers, but would allow the reuse of existing tools (at least for the behaviour, if not for the structural concepts). Such partial uses for our work are further discussed in the perspectives of Chapter 8.

The next chapter presents two case studies inspired from real-life.

# 7

# Case Studies

Our prototype, called Systems TEst LAnguage Environment (STELAE), was developed as an Eclipse plug-in for the Man-Machine Interface (MMI) component of the U-TEST Real-Time System [3]. STELAE allows us to illustrate the complete test development approach, from the editing of test models to the execution of the automatically generated code on the U-TEST Real-Time System test platform. The targeted executable test language was $PL_5$.

In this chapter we demonstrate the functionalities of STELAE on two simplified case studies inspired from real-life. They target Flight Warning System (FWS) (Section 7.1) and respectively Air Data Inertial Reference System (ADIRS) (Section 7.2) simplified models. We presented STELAE and the ADIRS case study in an article accepted at an international conference [135].

## 7.1 FWS - Engine Fire Alarm Synthesis

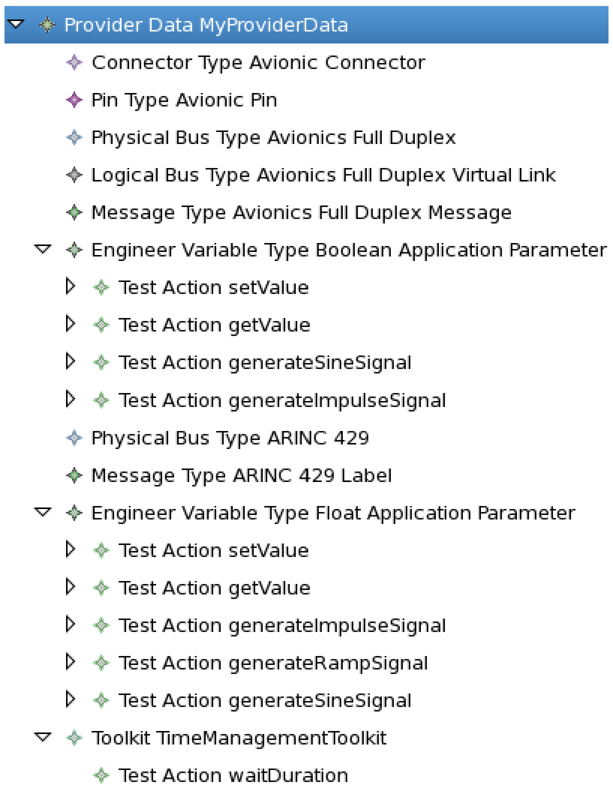
This first case study is inspired from a real one targeting an FWS. We chose this FWS-inspired test study as we had access to the real ICD. For confidentiality issues we cannot present the ICD or its model here. Our simplified FWS employs AFDX buses for transporting its input and output application parameters. We developed a simple simulation of the FWS behaviour in $PL_1$, based on the original detailed design. $PL_1$ is executable in parallel with $PL_5$, on the same target integration test platform: U-TEST Real-Time System [3].

An FWS monitors other systems of an aircraft, and in case failures or dangerous flight conditions are detected, informs the crew of these problems by means of alarms.

We deal here with the synthesis of an output alarm for an engine fire situation, based on four input partial alarms. This logic is validated in two steps. First the four input partial alarms are activated and the starting of the output alarm within 1 second is verified. Secondly, two among the four input partial alarms are deactivated and the stopping of the output alarm after 17 seconds is verified.

For this demonstration we defined a MyProviderData with the different types of SUT interface levels for an AFDX bus (Figure 7.1). Taking the logic of the test case into account, the test engineer is offered several test actions for the BooleanApplicationParameter type, such as: getValue() and setValue(). Test actions that do not correspond to interactions with the SUT are distributed inside toolkit structures. Such an example is waitDuration(), attached to the TimeManagementToolkit. Figure 7.1 does not show the parameters of these test actions, but they are part of the test model.

In STELAE, a password-based access control system restricts access to the test solution provider section.



```
Provider Data MyProviderData
    Connector Type Avionic Connector
    Pin Type Avionic Pin
    Physical Bus Type Avionics Full Duplex
    Logical Bus Type Avionics Full Duplex Virtual Link
    Message Type Avionics Full Duplex Message
    Engineer Variable Type Boolean Application Parameter
        Test Action setValue
        Test Action getValue
        Test Action generateSineSignal
        Test Action generateImpulseSignal
    Physical Bus Type ARINC 429
    Message Type ARINC 429 Label
    Engineer Variable Type Float Application Parameter
        Test Action setValue
        Test Action getValue
        Test Action generateImpulseSignal
        Test Action generateRampSignal
        Test Action generateSineSignal
    Toolkit TimeManagementToolkit
        Test Action waitDuration
```

**Figure 7.1: U-TEST MMI - STELAE Perspective - MyProviderData -**

With the user role, we entered the MyUserData part of the test model (Figure 7.2). Let us first look at the structural elements entered in the graphical editor on the left. We entered the FWS system under test and its interfaces, using the types available in MyProviderData. We defined an Alarm_reset_function test case that starts an instance of the STC_Alarm_reset_function sequential test component, in a new thread. The test component has its behaviour structured in four sequential blocks: Initialization, SetAlarmCheck, ResetAlarmCheck and DeInitialization. Initialization sets the four input partial alarms to false and waits for 1 second. SetAlarmCheck activates the four input partial alarms, waits for 1 second and then verifies that the output alarm is activated. ResetAlarmCheck deactivates two among the four input partial alarms, waits for 17 second and then verifies that the output alarm is deactivated. DeInitialization puts the FWS in a no-alarm state. Figure 7.2 also shows the test context for the other case study concerning the ADIRS, as well as another test case we implemented for the FWS but that we do not discuss here.

A number of Python scripts were generated. They can be noticed in the left “Model Project”view of Figure 7.2. After their execution in the “Console”view, the results of the test are shown in the “Test Management”view.

In order to test our implementation of the test verdict management, we inserted a fault inside the FWS model: the output alarm is not deactivated. As expected, the test verdict that was returned was `False`.

## 7.2 ADIRS - Consolidated Aircraft Speed Value

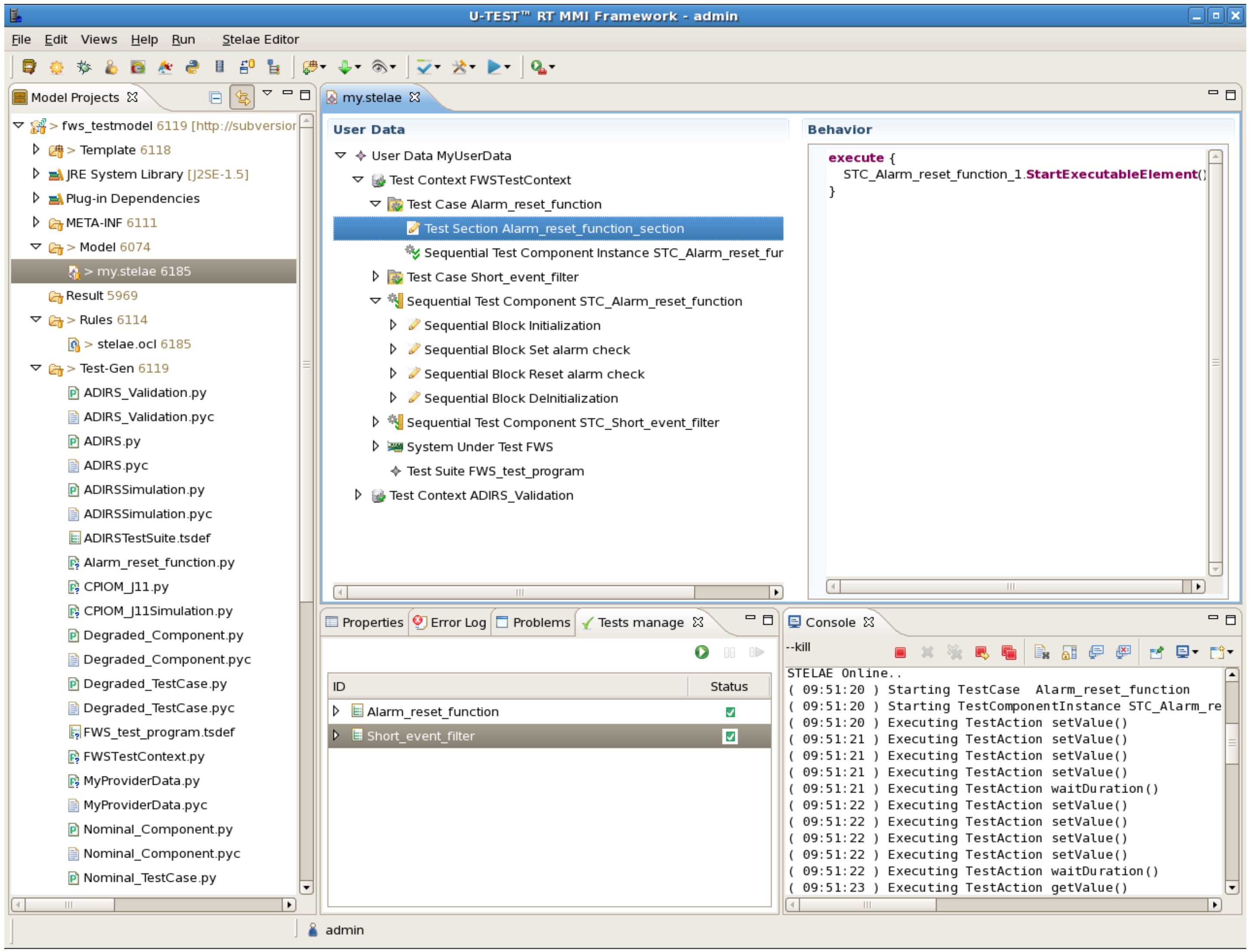
This case-study concerns the consolidated aircraft speed value functionality. It is inspired from a real one targeting the ADIRS [65]. We chose the ADIRS-inspired case study as it allowed us to demonstrate a number of domain-specific concepts that we integrated in the test meta-model (e.g., timed stimulations such as sine, the cycle-by-cycle test component). We did not have access to the real ICD, nor to the real ADIRS design.

We developed a simple simulation of part of the ADIRS behaviour in Python. The ADIRS deals with the acquisition of several application parameters necessary for the flight control system (e.g., altitude, speed, angle of attack). For each of these application parameters, redundant sensors exist and a consolidated value is computed from the set of available input values.

We deal here with the aircraft speed application parameter. The values of three input application parameters (AC_SPEED_1/2/3) are used to compute the value of the output consolidated application parameter (AC_SPEED). We assume that our simplified ADIRS employs ARINC 429 buses for transporting its input and output application parameters. Figure 7.3 shows the interfaces of the ADIRS in our test model.

The ADIRS logic is the following:

- **Nominal behaviour**: The consolidated value is the median of the three input values, if the median does not diverge from the other two values. The divergence is measured as the differences between the median value and the other two values. The median is divergent if these differences exceed a certain threshold during three consecutive execution cycles.



U-TEST™ RT MMI Framework - admin

File Edit Views Help Run Stelae Editor

Model Projects

```
▽ > fws_testmodel 6119 [http://subversior
  ▷ > Template 6118
  ▷ JRE System Library [J2SE-1.5]
  ▷ Plug-in Dependencies
  ▷ META-INF 6111
  ▽ > Model 6074
      > my.stelae 6185
    Result 5969
  ▽ > Rules 6114
      > stelae.ocl 6185
  ▽ > Test-Gen 6119
      ADIRS_Validation.py
      ADIRS_Validation.pyc
      ADIRS.py
      ADIRS.pyc
      ADIRSSimulation.py
      ADIRSSimulation.pyc
      ADIRSTestSuite.tsdef
      Alarm_reset_function.py
      CPIOM_J11.py
      CPIOM_J11Simulation.py
      Degraded_Component.py
      Degraded_Component.pyc
      Degraded_TestCase.py
      Degraded_TestCase.pyc
      FWS_test_program.tsdef
      FWSTestContext.py
      MyProviderData.py
      MyProviderData.pyc
      Nominal_Component.py
      Nominal_Component.pyc
      Nominal_TestCase.py
```

my.stelae

User Data

```
▽ User Data MyUserData
  ▽ Test Context FWSTestContext
    ▽ Test Case Alarm_reset_function
        Test Section Alarm_reset_function_section
        Sequential Test Component Instance STC_Alarm_reset_fur
    ▷ Test Case Short_event_filter
    ▽ Sequential Test Component STC_Alarm_reset_function
      ▷ Sequential Block Initialization
      ▷ Sequential Block Set alarm check
      ▷ Sequential Block Reset alarm check
      ▷ Sequential Block DeInitialization
    ▷ Sequential Test Component STC_Short_event_filter
    ▷ System Under Test FWS
      Test Suite FWS_test_program
  ▷ Test Context ADIRS_Validation
```

Behavior

```
execute {
  STC_Alarm_reset_function_1.StartExecutableElement()
}
```

Properties | Error Log | Problems | Tests manage

| ID | Status |
|---|---|
| Alarm_reset_function | ✓ |
| Short_event_filter | ✓ |

Console

--kill

```
STELAE Online..
( 09:51:20 ) Starting TestCase  Alarm_reset_function
( 09:51:20 ) Starting TestComponentInstance STC_Alarm_re
( 09:51:20 ) Executing TestAction setValue()
( 09:51:21 ) Executing TestAction setValue()
( 09:51:21 ) Executing TestAction setValue()
( 09:51:21 ) Executing TestAction setValue()
( 09:51:21 ) Executing TestAction waitDuration()
( 09:51:22 ) Executing TestAction setValue()
( 09:51:22 ) Executing TestAction setValue()
( 09:51:22 ) Executing TestAction setValue()
( 09:51:22 ) Executing TestAction setValue()
( 09:51:22 ) Executing TestAction waitDuration()
( 09:51:23 ) Executing TestAction getValue()
```

admin

Figure 7.2: U-TEST MMI - STELAE Perspective - FWS -

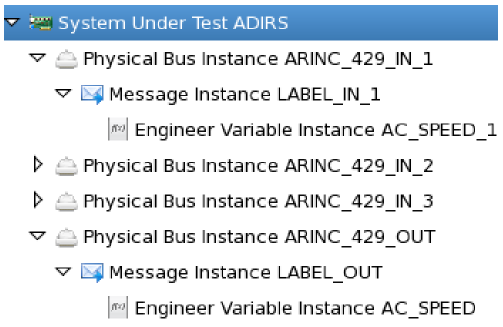

System Under Test ADIRS
Physical Bus Instance ARINC_429_IN_1
Message Instance LABEL_IN_1
Engineer Variable Instance AC_SPEED_1
Physical Bus Instance ARINC_429_IN_2
Physical Bus Instance ARINC_429_IN_3
Physical Bus Instance ARINC_429_OUT
Message Instance LABEL_OUT
Engineer Variable Instance AC_SPEED

**Figure 7.3: ADIRS - SystemUnderTest -**

- **Degraded behaviour**: If the median of the three input values diverges from the remaining two for more than three cycles, then the source having produced the median is permanently eliminated. The consolidated value is the average of the remaining two values.

For this system under test, we consider here two test cases verifying the behaviour:

- **Nominal behaviour test case**: Verify that the consolidated value remains equal to the median of the three input values in the presence of a small-amplitude sine oscillation that does not render the three input values divergent.

- **Degraded behaviour test case**: Inject a divergence on one of the three input values and verify that the consolidated value is equal to the average of the two remaining values. Verify that the divergent source is permanently eliminated, even if the divergence is corrected.

These test cases should be executed on all combinations of input application parameters. For example, one can render divergent first AC_SPEED_1, then AC_SPEED_2 and finally AC_SPEED_3. As the tested behaviour is identical in all these three cases, the reuse of a parametrized test component is convenient.

We first discuss the functionalities that the test solution provider must make available to test engineers (Figure 7.1). Let us assume that only the following elements are already available to the test engineer: the AFDX bus and float application parameter types. The following predefined test actions are accessible for the FloatApplicationParameter type: setValue(), getValue() and generateRampSignal(). But the test engineer will also need access to the ARINC 429 bus type and to the generateSineSignal() test action on float application parameter type. Consequently, the test solution provider can add them to the list of already existing ones using the predefined extension points. Once these elements are rendered available by the test solution provider, they can be used by the test engineers in order to model the ICD of their SUT and to call test actions on its different interface elements.

For our case study, a test engineer would define a unique ADIRS_Validation test context, comprising the two test cases: Nominal_TestCase and Degraded_TestCase. Two test components are added as well to the test context: Nominal_Component and Degraded_Component. In the framework of the case study, each test component is instantiated once within each previously mentioned test cases: Nominal_Component_1 and respectively Degraded_Component_1. In real life, there would be several test cases, each with its own component instances.

In order to render the test components reusable, we add four formal interfaces to each one: three for the inputs (First_IN, Second_IN and Third_IN) and one for the output (OUT). The connection to the corresponding permutation of ADIRS parameters is defined within the test architectures owned by the test cases.

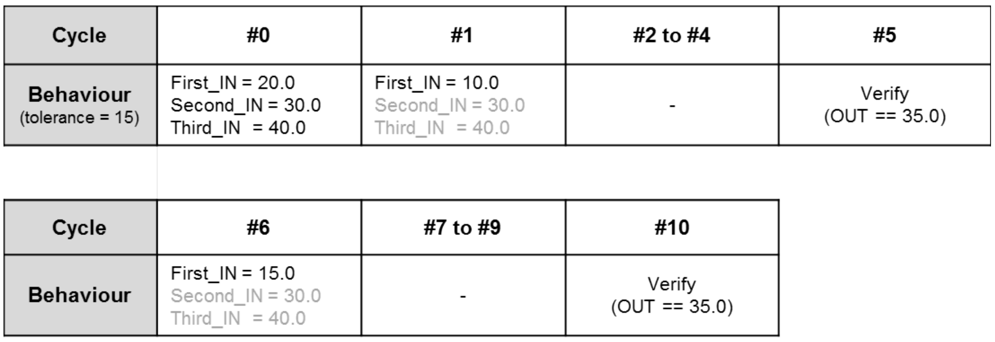
Let us now look at our two test components. First it is important to mention that the two test components are of different types, as they need to behave differently in order to validate the SUT. The Nominal_TestComponent is a sequential test component because the test logic is fairly simple, while the Degraded_Component is a cycle-by-cycle test component, as a finer temporal control was needed for expressing the tolerance in number of cycles the ADIRS has with regard to its inputs.

We defined three sequential blocks for the Nominal_Component sequential test component: Initialization, Stimulation and Behaviour. The Initialization sequential block initializes the SUT by setting three coherent values for the three input engineer variables (First/Second/Third_IN). Notice that we refer here to the formal interfaces of the test component. The Stimulation sequential block applies a sine signal on one of the input engineer variables (Second_IN). The sine signal does not render the engineer variable divergent with regard to the remaining two. The Behaviour sequential block verifies that the value for the output parameter (OUT) is the median.

Figure 7.4 exemplifies the cycle-by-cycle behaviour for the Degraded_Component test component. First the ADIRS is initialized with coherent values for the three input engineer variables. Next, one of the inputs (First_IN) is rendered divergent and the fact that the divergent source has been eliminated after three cycles is verified. Finally the divergent source is rendered coherent and the fact that it remains permanently eliminated is verified. Note that in our case study the chosen values for the aircraft speed are fictional and do not correspond to realistic ones.

Figure 7.5 shows the corresponding model in the STELAE graphical editor.

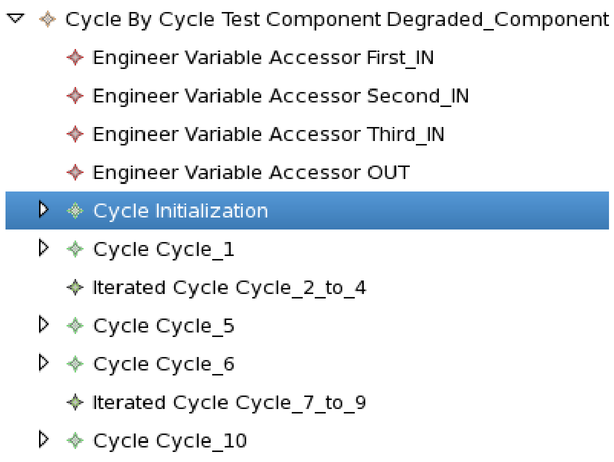
Listings 7.1 and 7.2 show the behaviour of the Initialization and respectively Cycle_5 cycles in the STELAE textual editor. It is important to re-mention that the concrete syntax presented is only an example, as several ones can be defined for the test meta-model, catering to the individual needs and tastes of the different users.



| Cycle | #0 | #1 | #2 to #4 | #5 |
|---|---|---|---|---|
| **Behaviour** (tolerance = 15) | First_IN = 20.0<br>Second_IN = 30.0<br>Third_IN = 40.0 | First_IN = 10.0<br>Second_IN = 30.0<br>Third_IN = 40.0 | - | Verify<br>(OUT == 35.0) |

| Cycle | #6 | #7 to #9 | #10 |
|---|---|---|---|
| **Behaviour** | First_IN = 15.0<br>Second_IN = 30.0<br>Third_IN = 40.0 | - | Verify<br>(OUT == 35.0) |

Figure 7.4: Degraded Component - Behaviour Description -



```
Cycle By Cycle Test Component Degraded_Component
    Engineer Variable Accessor First_IN
    Engineer Variable Accessor Second_IN
    Engineer Variable Accessor Third_IN
    Engineer Variable Accessor OUT
    Cycle Initialization
    Cycle Cycle_1
    Iterated Cycle Cycle_2_to_4
    Cycle Cycle_5
    Cycle Cycle_6
    Iterated Cycle Cycle_7_to_9
    Cycle Cycle_10
```

Figure 7.5: Degraded Component - Low-Level Structural Elements -

```
CallAccessor  first_IN.setValue(20.0)
CallAccessor  second_IN.setValue(30.0)
CallAccessor  third_IN.setValue(40.0)
```

**Listing 7.1:** Degraded Component - Initialization Cycle

```
Check (OUT == 35.0)
```

**Listing 7.2:** Degraded Component - Cycle_5 Cycle

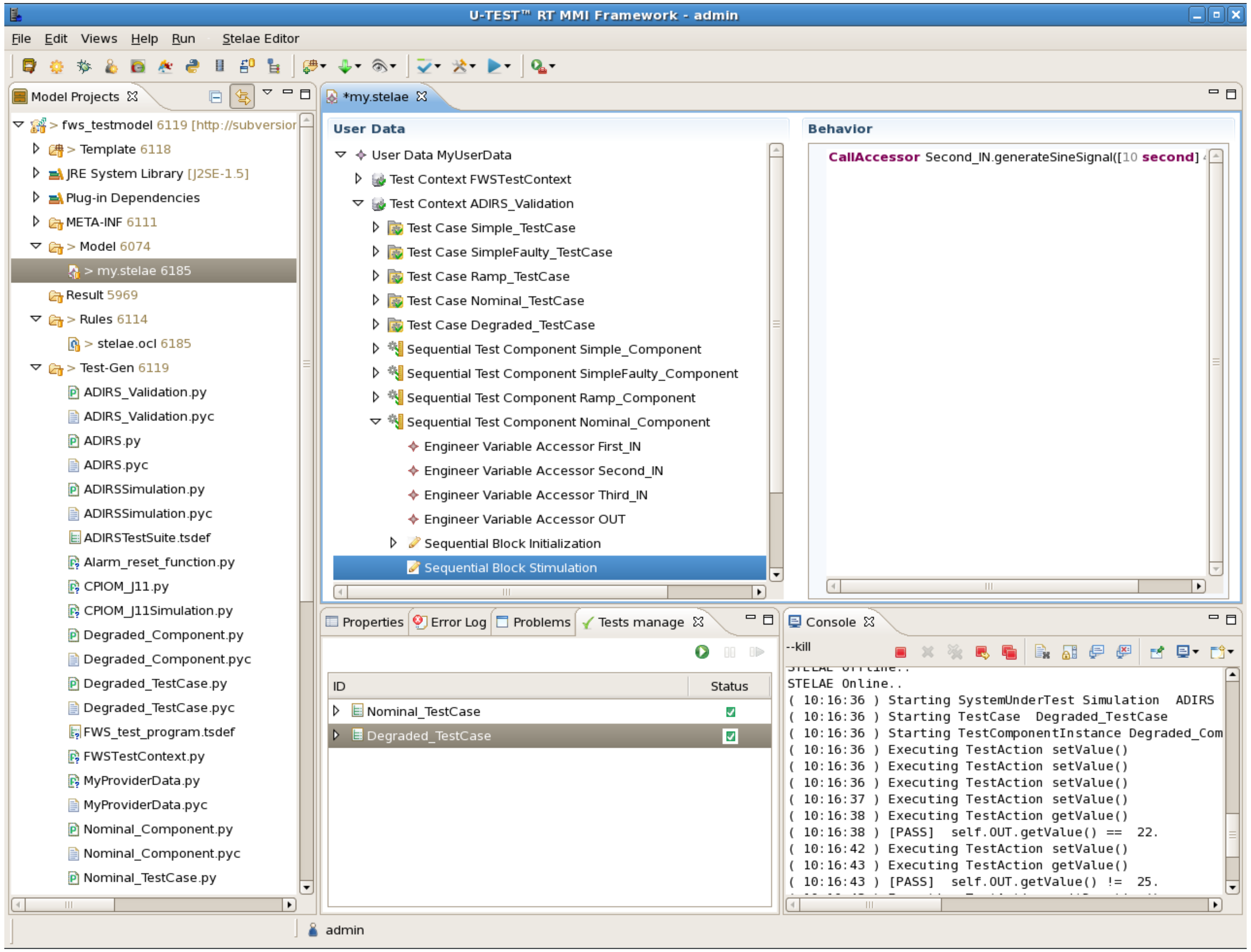
In order to test one of our OCL checks and the `Error` verdict, we inserted a fault inside one of the cycles of the Degraded_Component: a waitDuration() call with a parameter higher than the cycle duration. The OCL check we defined for such situations correctly identified the design problem. Nevertheless, we pursued with the production of the erroneous code. The problem was also identified at runtime, when our implementation set an `Error` verdict as the duration of the cycles was not being respected during execution.

As can be seen in Figure 7.6, we implemented several other test cases for the ADIRS in order to test our approach. We do not discuss those here as they do not bring new information in comparison to the test cases we already presented. Figure 7.6 also shows the test context for the other case study concerning the FWS.

For the Nominal_TestCase, Figure 7.7 shows the “Runtime”perspective of the U-TEST MMI component, where we can observe the modification of the values for our different application parameters during execution. The “Array”view on the left comprises a list of application parameters that we observe during the execution of the tests. Notice the three AC_SPEED_1/2/3_STATUS variables. They are internal to our simulation of the ADIRS (i.e., not part of the ICD), corresponding to whether a source was eliminated or not because of its divergence from the other two. We rendered them observable in order to see the internal state of the simulated SUT. The different “Oscilloscope”views show the evolution of the different engineer variables in time. Notice the sine variation injected by the Degraded_TestCase.

## 7.3 Conclusion

In this chapter we demonstrated the functionalities of our first prototype called STELAE on two case studies inspired from real-life. They concerned the FWS and ADIRS avionics embedded systems. For each of these systems we developed a simulation for their behaviour, for the first one in $PL_1$ and for the second one in Python. We had access to the real ICD of the FWS. For the ADIRS we defined one from scratch. We focused on the AFDX avionic bus in the first test case, while the second employed the ARINC 429 avionic bus.



```
fws_testmodel 6119 [http://subversior
  Template 6118
  JRE System Library [J2SE-1.5]
  Plug-in Dependencies
  META-INF 6111
  Model 6074
    my.stelae 6185
  Result 5969
  Rules 6114
    stelae.ocl 6185
  Test-Gen 6119
    ADIRS_Validation.py
    ADIRS_Validation.pyc
    ADIRS.py
    ADIRS.pyc
    ADIRSSimulation.py
    ADIRSSimulation.pyc
    ADIRSTestSuite.tsdef
    Alarm_reset_function.py
    CPIOM_J11.py
    CPIOM_J11Simulation.py
    Degraded_Component.py
    Degraded_Component.pyc
    Degraded_TestCase.py
    Degraded_TestCase.pyc
    FWS_test_program.tsdef
    FWSTestContext.py
    MyProviderData.py
    MyProviderData.pyc
    Nominal_Component.py
    Nominal_Component.pyc
    Nominal_TestCase.py
```

User Data

```
User Data MyUserData
  Test Context FWSTestContext
  Test Context ADIRS_Validation
    Test Case Simple_TestCase
    Test Case SimpleFaulty_TestCase
    Test Case Ramp_TestCase
    Test Case Nominal_TestCase
    Test Case Degraded_TestCase
    Sequential Test Component Simple_Component
    Sequential Test Component SimpleFaulty_Component
    Sequential Test Component Ramp_Component
    Sequential Test Component Nominal_Component
      Engineer Variable Accessor First_IN
      Engineer Variable Accessor Second_IN
      Engineer Variable Accessor Third_IN
      Engineer Variable Accessor OUT
      Sequential Block Initialization
      Sequential Block Stimulation
```

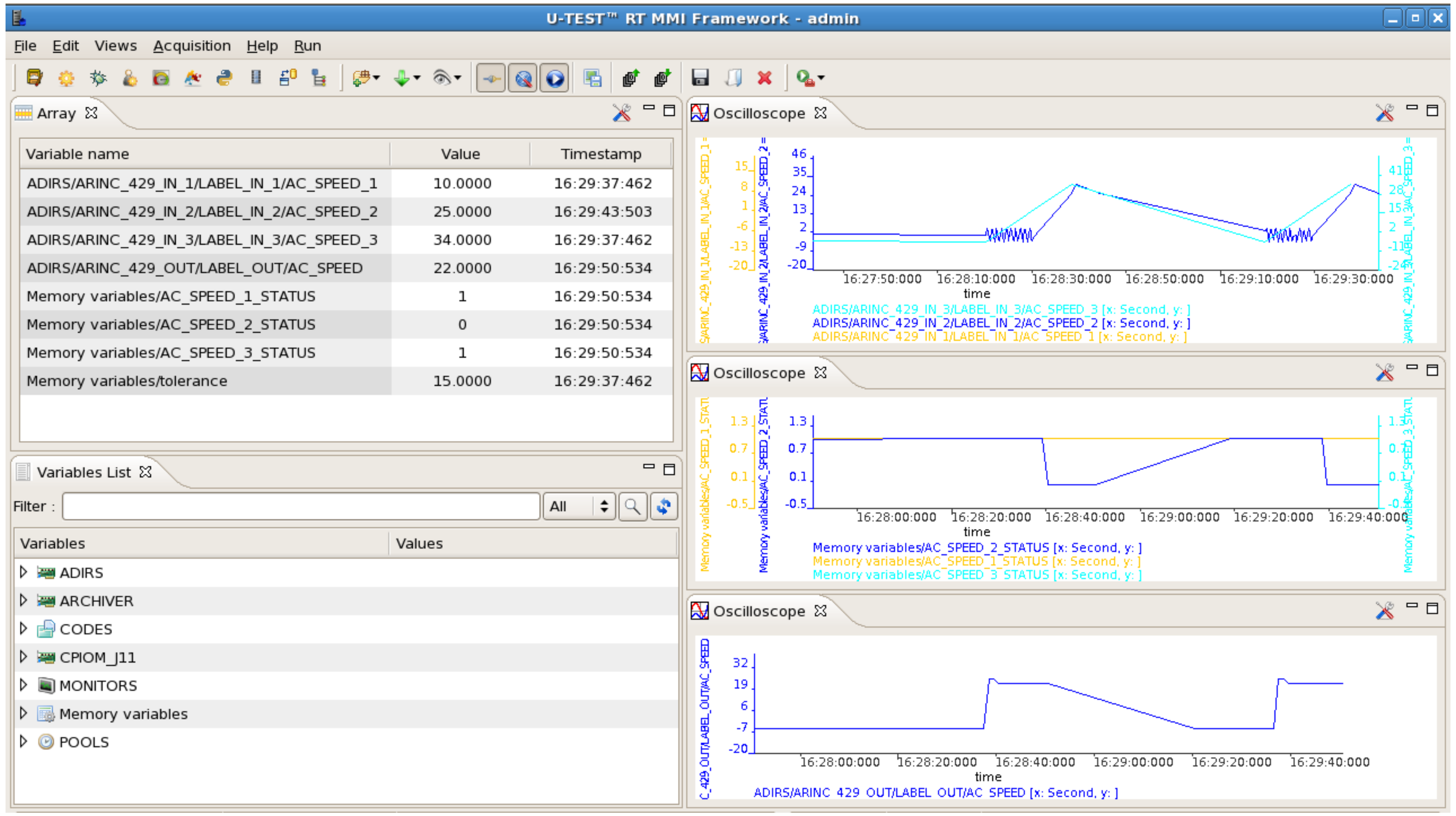
Behavior

```
CallAccessor Second_IN.generateSineSignal([10 second]
```

| ID | Status |
|---|---|
| Nominal_TestCase | ✓ |
| Degraded_TestCase | ✓ |

```
--kill
STELAE Online..
( 10:16:36 ) Starting SystemUnderTest Simulation  ADIRS
( 10:16:36 ) Starting TestCase  Degraded_TestCase
( 10:16:36 ) Starting TestComponentInstance Degraded_Com
( 10:16:36 ) Executing TestAction setValue()
( 10:16:36 ) Executing TestAction setValue()
( 10:16:36 ) Executing TestAction setValue()
( 10:16:37 ) Executing TestAction setValue()
( 10:16:38 ) Executing TestAction getValue()
( 10:16:38 ) [PASS]  self.OUT.getValue() ==  22.
( 10:16:42 ) Executing TestAction setValue()
( 10:16:43 ) Executing TestAction getValue()
( 10:16:43 ) [PASS]  self.OUT.getValue() !=  25.
```

Figure 7.6: U-TEST MMI - STELAE Perspective - ADIRS -


U-TEST™ RT MMI Framework - admin
File Edit Views Acquisition Help Run
Array
Oscilloscope
Variables List
Filter :
All
time
ADIRS/ARINC_429_IN_3/LABEL_IN_3/AC_SPEED_3 [x: Second, y: ]
ADIRS/ARINC_429_IN_2/LABEL_IN_2/AC_SPEED_2 [x: Second, y: ]
ADIRS/ARINC_429_IN_1/LABEL_IN_1/AC_SPEED_1 [x: Second, y: ]
Memory variables/AC_SPEED_2_STATUS [x: Second, y: ]
Memory variables/AC_SPEED_1_STATUS [x: Second, y: ]
Memory variables/AC_SPEED_3_STATUS [x: Second, y: ]
ADIRS/ARINC_429_OUT/LABEL_OUT/AC_SPEED [x: Second, y: ]

| Variable name | Value | Timestamp |
|---|---|---|
| ADIRS/ARINC_429_IN_1/LABEL_IN_1/AC_SPEED_1 | 10.0000 | 16:29:37:462 |
| ADIRS/ARINC_429_IN_2/LABEL_IN_2/AC_SPEED_2 | 25.0000 | 16:29:43:503 |
| ADIRS/ARINC_429_IN_3/LABEL_IN_3/AC_SPEED_3 | 34.0000 | 16:29:37:462 |
| ADIRS/ARINC_429_OUT/LABEL_OUT/AC_SPEED | 22.0000 | 16:29:50:534 |
| Memory variables/AC_SPEED_1_STATUS | 1 | 16:29:50:534 |
| Memory variables/AC_SPEED_2_STATUS | 0 | 16:29:50:534 |
| Memory variables/AC_SPEED_3_STATUS | 1 | 16:29:50:534 |
| Memory variables/tolerance | 15.0000 | 16:29:37:462 |

| Variables | Values |
|---|---|
| ADIRS | |
| ARCHIVER | |
| CODES | |
| CPIOM_J11 | |
| MONITORS | |
| Memory variables | |
| POOLS | |

Figure 7.7: U-TEST MMI - Runtime Perspective - ADIRS -

We covered a number of test engineer activities on these case studies: the definition of test models, the automatic generation of files and code, as well as the execution on top of a real integration test platform: the U-TEST Real-Time System [3]. Currently we can demonstrate these functionalities on simple to medium complexity test cases.

Once the SUT interfaces are entered, defining the different test models of our case studies took only a couple of minutes.

One of our perspectives is to demonstrate the functionalities of STELAE on more complex case studies: attacking different ICD hierarchical levels for communication with the SUT, as well as targeting richer behaviour with a more complicated temporal logic. Another perspective would be to define a benchmark comprising a number of test case specifications, and afterwards compare the experience of test engineers with STELAE against their experience with existing test language. We could measure the time they spend for defining the test cases, the number of interactions with the user interfaces, as well the number of errors. This would allow us to assess in an objective manner the gain that our approach may deliver in comparison with existing ones.

8

# Conclusion and Perspectives

This project investigated the definition of a model-driven engineering approach for the development of in-the-loop tests for avionics embedded systems. We were motivated by the fact that existing test development solutions used in our field no longer respond to the needs expressed by the stakeholders: aircraft manufacturers, equipment/system providers and test solution providers. These evolving requirements concern the capability to exchange portable tests between stakeholders, to more easily customize and maintain the test development solution, to increase the productivity of test engineers and the quality of the tests. In addition, no standard exists or is emerging in our field, in contrast to other fields. For example, hardware testing at the production and maintenance life-cycle phases benefits from the ATLAS [112] and ATML [113] international standards. Unfortunately, existing standards such as those previously mentioned are not directly reusable in our industrial context, as they do not offer the needed specific functionalities. Finally, test engineers do not benefit from the latest methodologies and technologies issued from software engineering, such as model-driven engineering. In order to tackle all these issues, we propose a model-driven approach in which test models take the central position of the test development activity, replacing the test code that currently holds this position. A meta-model is the foundation of any model-driven approach, as it constraints the definition of test models in a manner similar to that in which the grammar of a programming language constraints the writing of code. A meta-model integrates the domain-specific concepts and their relations. Consequently, in order to define our own test meta-model we needed to pursue the analysis of current practice.

Our first contribution consists in the analysis of a sample of four proprietary test languages currently used in our industrial domain. For comparison purposes, we also chose two test languages outside the field of avionics: TestML [114], that was issued from a research project in the automotive domain, and TTCN-3 [82], which is a mature international standard used for the testing of telecommunications protocols and distributed systems. To the best of our knowledge, no such analysis has been previously performed. We focus on four major categories of features exhibited by the test languages:

- the organization of the tests,
- the abstraction and access to the interfaces of the SUT,
- the language instructions for performing test control and test interaction actions,
- the management of time.

Our analysis followed a clear methodology. For each feature that was identified we looked at the way it was offered by the test languages. This was based on the analysis of different available information for each test language, such as: user manuals, presentations, articles and books, as well as dedicated test specifications and procedures. In case a test language did not offer one feature or offered it in a manner different from the other test languages in our sample, we discussed the issue with test engineers, asking for clarifications. This analysis allowed us to identify a rich set of domain-specific features/concepts, together with best practices and pitfalls to avoid. The analysis confirmed the initial hypotheses that motivated our work: the world of test languages in our industrial context is heterogeneous, but it was possible to identify common concepts. The conclusions of our test languages analysis were summarized in a list of meta-modelling guiding principles that steered the definition of our test meta-model. In addition to representing the foundation for our meta-modelling work, the results of this first contribution are immediately useful to stakeholders in this field, allowing them to identify manners in which they could improve existing test languages or define new ones from scratch.

The second contribution consists in the definition of a test meta-model that integrates the rich set of domain-specific concepts in a homogeneous manner, while taking into account the best practices in the field. The test meta-model separates test solution provider and test solution user concerns. Consequently it allows the test solution provider to more easily customize and maintain the test solution. The test meta-model also separates structural aspects from behavioural ones. This separation was in favour of developing a mixed editor, where structural elements are entered graphically and behavioural ones coded in a textual form. In spite of this separation, structural and behavioural elements are consistently integrated inside the test meta-model, with constraints on the type of behaviour specifiable for a structural element. Such constraints are intended to serve fault avoidance purpose. In addition, the test meta-model proposes strongly-typed elements for the SUT interfaces and attached test actions. We used meta-modelling not only as an instrument in order to define our domain-specific concepts and their relations, but also to gain access to a rich set of existing free and open-source tools: graphical and textual model editors, checkers and automatic code generators. This enabled us to rapidly develop our first prototype.

Our third contribution consisted in the implementation of a first prototype on top of a real integration test platform: the commercially available U-TEST Real-Time System [3] developed at Cassidian Test & Services. We called the prototype: Systems TEst LAnguage Environment (STELAE). We used a template-based model-to-text transformation, targeting a Python-based executable proprietary test language. The target test language did not exist at the time of our first analysis of the field and consequently provided us the opportunity to challenge the genericity of our test meta-model, as well as of our test languages analysis. We proposed a method for organising the automatic code generation templates and the automatically generated code and files, in order to simplify the implementation task, based on the organisation of the test meta-model. We did not consider more advanced transformation functionalities, the template-based approach being sufficient for our needs. For the development of tests by test engineers, STELAE integrates a mixed (graphical and textual) customizable test development environment that was plugged in the Man-Machine Interface (MMI) component of the U-TEST Real-Time System. Finally, this prototype allowed us to evaluate the employed technology with regard to possible future industrialization. Our meta-model stood the test of implementation, with very limited modifications being brought to it during this phase of our project.

Our STELAE prototype was employed for the modelling of two case studies inspired from real-life. We can currently demonstrate the definition of test models, their implementation and execution, for simple to medium complexity test cases.

A first perspective of our work consists in expanding the sample of analysed test languages with test languages used in other domains, in order to evaluate the possibility of having a unique homogeneous solution between different industrial fields (e.g., between the avionics and automotive industries).

A second perspective is on the subject of the test meta-model semantics. Currently, most of the static semantics is found inside the test meta-model, with the rest being distributed inside OCL rules, Java classes, informal textual documents, and the textual representation. A formal semantics would be desirable for the test meta-model, especially for its execution dynamics. A major issue is that executable test languages targeted for the implementation of test models do not have a formal semantics themselves. The same problem arises for TTCN-3. In addition, a single semantics seems difficult to attach to the test meta-model, because of the different variation points. For example, the semantics of a periodic component in a hardware-in-the-loop situation would be different from that of a same test component type used in a model-in-the-loop situation, where a logical view of execution cycles suffices.

Another perspective would be to finalise the implementation of the test model development environment with more user-friendly functionalities needed in a production environment, while alleviating the multi-editor synchronization issues we identified. The automation of currently manual activities in our prototype, such as the definition of SUT interfaces by parsing an ICD, would be a welcome addition. Moreover, wizards guiding test engineers and allowing them access only to restricted portions on the test meta-model, based on their choices, could simplify the user experience and hide the complexity of the test meta-model.

A fourth important perspective deals with the investigation of better debugging functionalities. This problem can be analyzed from two points of view: whether it concerns the debugging performed by the test solution provider on the automatic code generation templates, or the debugging performed by the test engineer on the test model.

Regarding faults within automatic code generation templates, our work seeked to alleviate the problem. Wrappers were defined around target test language instructions, in order to enhance the clarity of the link between the automatically generated code and the textual representation of the test model behaviour. This direction can further be explored. An agile approach based on use-cases (i.e., exemplary input models and their expected output files/code) guided our definition of the automatic code generation templates. The use-cases served for the verification of the transformation. Finally we proposed an organization of the automatic code generation templates and automatically generated files/code based on the structure of the test meta-model, that eases their maintenance by engineers accustomed to the test meta-model.

Regarding faults at test model level, three approaches can be envisioned. The first approach consists in tackling the debugging issue at test model level, for example by inserting the breakpoint concept inside the test meta-model, employing annotation-based round-trip engineering techniques or developing specific debuggers. The second solution consists in bringing closer together the grammars of the textual representation of the test meta-model behavioural part and of the target test language. The concrete syntax for the test model behaviour would nevertheless be constrained by the test meta-model: although the concrete syntax of STELAE resembles that of Python, they are not identical. The third approach is more drastic, taking advantage of the modularity of the test meta-model to replace its behavioural part with the target test language. In this case, existing debugging tools could be reused (for the behavioural part), but the fault avoidance, customizability and portability characteristics of our approach would be forfeited. The already defined automatic code generation templates that target structural concepts could be reused as well. The used implementation language would nevertheless need to be upgraded with functionalities offered by the test meta-model (e.g., time-bounded structures, constraints on behaviour depending on the structure).

This is an important issue to be analysed on its own because of the complexity and specificities of the test meta-model. On-the-fly automatic code generation would be necessary in order to gain access inside the test language to structural elements defined in the test model. We consider that a more thorough evaluation of these issues is required.

Other partial uses for our work consist in the upgrade of existing test languages with concepts and best practices issued from our test languages analysis/test meta-model. Inside Cassidian Test & Services, on-going discussions concern the upgrade of $PL_5$ with test components and a richer verdict management, together with the upgrade of the U-TEST Real-Time System internal ICD representation to take into account strongly-typed constructs, with a list of test actions attached to SUT interface types. The graphical editor could also serve as a basis for the generation of richer test case skeletons than those currently offered, improving the user experience with $PL_1$ or $PL_5$.

Finally, the linking of our approach with existing model-based testing (for test generation from SUT functional specification) techniques can be envisioned. Abstract test cases derived from the SUT behaviour specification could be translated into our test models, for implementation purposes.

# References

[1] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004. 1, 5

[2] G. Bartley and B. Lingberg. Certification concerns of integrated modular avionics (ima) systems. In *2008 IEEE/AIAA 27$^{th}$ Digital Avionics Systems Conference*, DASC '08, pages 1.E.1–1–1.E.1–12, oct. 2008. doi: 10.1109/DASC.2008.4702766. 1, 19, 26

[3] U-test real-time system. URL `http://www.eads-ts.com/web/products/software/utest.html`. 3, 30, 89, 93, 119, 123, 133, 137

[4] R. G. Ebenau, S. H. Strauss, and S. S. Strauss. *Software Inspection Process*. McGraw-Hill Systems Design & Implementation Series. McGraw-Hill, 1994. ISBN 9780070621664. URL `http://books.google.fr/books?id=kSYiAQAAIAAJ`. 5

[5] P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 138–156, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41635-8. URL `http://dl.acm.org/citation.cfm?id=647348.724445`. 5

[6] P. Cuoq, F. Kirchner, N. Kosmatov, V. Prevosto, J. Signoles, and B. Yakobowski. Frama-c - a software analysis perspective. In *SEFM*, pages 233–247, 2012. 5

[7] P. Schnoebelen. *Vérification de logiciels: Techniques et outils du model-checking*. Vuibert informatique. Vuibert, 1999. ISBN 9782711786466. URL `http://books.google.fr/books?id=u0-_PAAACAAJ`. 5

[8] O. Grumberg and H. Veith. *25 Years of Model Checking: History, Achievements, Perspectives*. Springer Publishing Company, Incorporated, 1 edition, 2008. ISBN 3540698493, 9783540698494. 5

[9] J. F. Monin. *Comprendre les méthodes formelles: Panorama et outils logiques*. Collection technique et scientifique des télécommunications. Masson, 1996. ISBN 9782225853043. URL `http://books.google.fr/books?id=kLApAQAACAAJ`. 5

[10] A. Robinson and A. Voronkov, editors. *Handbook of Automated Reasoning*. Elsevier Science Publishers B. V., Amsterdam, The Netherlands, The Netherlands, 2001. ISBN 0-444-50812-0. 5

[11] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using formal specifications to support testing. *ACM Computing Survey*, 41(2):9:1–9:76, 2009. ISSN 0360-0300. doi: 10.1145/1459352.1459354. URL `http://doi.acm.org/10.1145/1459352.1459354`. 5

[12] IEEE Standard Glossary of Software Engineering Terminology. Technical report, 1990. URL `http://dx.doi.org/10.1109/IEEESTD.1990.101064`. 5, 7, 8

[13] H. Waeselynck P. Thévenod-Fosse and Y. Crouzet. Software statistical testing. *Predictably Dependable Computing Systems*, ESPRIT Basic Research Series:253–72, 1995. 6

[14] S.-D. Gouraud, A. Denise, M.-C. Gaudel, and B. Marre. A new way of automating statistical testing methods. In *Proceedings of the 2001 IEEE 16$^{th}$ International Conference on Automated Software Engineering*, ASE '01, pages 5–, Washington, DC, USA, 2001. IEEE Computer Society. URL `http://dl.acm.org/citation.cfm?id=872023.872550`. 6

[15] M. Petit and A. Gotlieb. Uniform selection of feasible paths as a stochastic constraint problem. In *Proceedings of the 7$^{th}$ International Conference on Quality Software*, QSIC '07, pages 280–285, Washington, DC, USA, 2007. IEEE Computer Society. ISBN 0-7695-3035-4. URL `http://dl.acm.org/citation.cfm?id=1318471.1318535`. 6

[16] S. Poulding and J. A. Clark. Efficient software verification: Statistical testing using automated search. *IEEE Transactions of Software Engineering*, 36(6):763–777, 2010. ISSN 0098-5589. doi: 10.1109/TSE.2010.24. URL `http://dx.doi.org/10.1109/TSE.2010.24`. 6

[17] R. A. DeMillo, R. J. Lipton, and F. G. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4): 34–41, 1978. ISSN 0018-9162. doi: 10.1109/

C-M.1978.218136. URL http://dx.doi.org/10.1109/C-M.1978.218136. 7

[18] M. Daran and P. Thévenod-Fosse. Software error analysis: a real case study involving real faults and mutations. In *Proceedings of the 1996 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '96, pages 158–171, New York, NY, USA, 1996. ACM. ISBN 0-89791-787-1. doi: 10.1145/229000.226313. URL http://doi.acm.org/10.1145/229000.226313. 7

[19] A. J. Offutt and R. H. Untch. Mutation 2000: Uniting the orthogonal. In W. Eric Wong, editor, *Mutation Testing for the New Century*, pages 34–44. Kluwer Academic Publishers, Norwell, MA, USA, 2001. ISBN 0-7923-7323-5. URL http://dl.acm.org/citation.cfm?id=571305.571314. 7

[20] B. Beizer. *Software Testing Techniques (2$^{nd}$ Edition)*. Van Nostrand Reinhold Co., New York, NY, USA, 1990. ISBN 0-442-20672-0. 7

[21] G. J. Myers and C. Sandler. *The Art of Software Testing, Second Edition*. John Wiley & Sons, 2004. ISBN 0471469122. 7

[22] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008. ISBN 9780521880381. URL http://books.google.fr/books?id=BMbaAAAAMAAJ. 7

[23] B. Legeard, F. Bouquet, and N. Pickaert. *Industrialiser le test fonctionnel. Pour maîtriser les risques métier et accroître l'efficacité du test*. Collection InfoPro. Dunod, 2011. URL http://hal.inria.fr/hal-00645019. 2ème édition. 304 pages. ISBN : 9782100566563. 7

[24] S.C. Ntafos. A comparison of some structural testing strategies. *IEEE Transactions on Software Engineering*, 14(6):868–874, 1988. ISSN 0098-5589. doi: http://doi.ieeecomputersociety.org/10.1109/32.6165. 7

[25] S. Rapps and E. J. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, 11(4):367–375, 1985. ISSN 0098-5589. doi: 10.1109/TSE.1985.232226. URL http://dx.doi.org/10.1109/TSE.1985.232226. 7

[26] J.J. Chilenski and S.P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9 (5):193–200, sep 1994. ISSN 0268-6961. 7

[27] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson. A practical tutorial on modified condition/decision coverage - nasa technical report. Technical report, 2001. 7

[28] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Communications of the ACM*, 31(6):676–686, June 1988. ISSN 0001-0782. doi: 10.1145/62959.62964. URL http://doi.acm.org/10.1145/62959.62964. 8, 15

[29] M. Grindal, B. Lindström, J. Offutt, and S. F. Andler. An evaluation of combination strategies for test case selection. *Empirical Software Engineering*, 11(4):583–611, 2006. ISSN 1382-3256. doi: 10.1007/s10664-006-9024-2. URL http://dx.doi.org/10.1007/s10664-006-9024-2. 8

[30] J. Dick and A. Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *Proceedings of the 1$^{st}$ International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods*, FME '93, pages 268–284, London, UK, 1993. Springer-Verlag. ISBN 3-540-56662-7. URL http://dl.acm.org/citation.cfm?id=647535.729387. 9

[31] R. M. Hierons. Testing from a z specification. In *Journal of Software Testing, Verification and Reliability*, volume 7, pages 19–33, London, UK, UK, 1997. 9

[32] L. Van Aertryck, M. V. Benveniste, and D. Le Mtayer. Casting: A formally based software test generation method. In *Proceedings of the 1997 IEEE 1$^{st}$ International Conference on Formal Engineering Methods*, ICFEM '97, pages 101–, 1997. URL http://dblp.uni-trier.de/db/conf/icfem/icfem1997.html#AertryckBM97. 9

[33] S. Behnia and H. Waeselynck. Test criteria definition for b models. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems*, volume 1 of *FM '99*, pages 509–529, London, UK, 1999. Springer-Verlag. ISBN 3-540-66587-0. URL http://dl.acm.org/citation.cfm?id=647544.730455. 9

[34] B. Legeard, F. Peureux, and M. Utting. Controlling test case explosion in test generation from b formal models: Research articles. *Software Testing, Verification and Reliability*, 14 (2):81–103, 2004. ISSN 0960-0833. doi: 10.1002/stvr.v14:2. URL http://dx.doi.org/10.1002/stvr.v14:2. 9

[35] A. J. Offutt, S. Liu, A. Abdurazik, and P. Ammann. Generating test data from state-based specifications. *Software Testing, Verification and Reliability*, 13(1):25–53, 2003. URL http://dblp.uni-trier.de/db/journals/stvr/stvr13.html#OffuttLAA03. 9

[36] J. C. Huang. An approach to program testing. *ACM Computing Survey*, 7(3):113–128, 1975. ISSN 0360-0300. doi: 10.1145/356651.356652. URL `http://doi.acm.org/10.1145/356651.356652`. 9

[37] Tsunoyama M. Naito, S. Fault detection for sequential machines by transition tours. *Procedings of the 1981 IEEE Fault Tolerant Computing Symposium*, pages 238–243, 1981. 9

[38] J. C. Rault. An approach towards reliable software. In *Proceedings of the 4th International Conference on Software Engineering*, ICSE '79, pages 220–230, Piscataway, NJ, USA, 1979. IEEE Press. URL `http://dl.acm.org/citation.cfm?id=800091.802942`. 9

[39] T. S. Chow. Testing software design modeled by finite-state machines. *IEEE Transactions on Software Engineering*, 4(3):178–187, May 1978. ISSN 0098-5589. doi: 10.1109/TSE.1978.231496. URL `http://dx.doi.org/10.1109/TSE.1978.231496`. 9

[40] K. Sabnani and A. Dahbura. A new technique for generating protocol test. *SIGCOMM Comp utuer Communication Review*, 15(4):36–43, 1985. ISSN 0146-4833. doi: 10.1145/318951.319003. URL `http://doi.acm.org/10.1145/318951.319003`. 9

[41] H. Ural. Formal methods for test sequence generation. *Computer Communications*, 15(5):311–325, 1992. ISSN 0140-3664. doi: 10.1016/0140-3664(92)90092-S. URL `http://dx.doi.org/10.1016/0140-3664(92)90092-S`. 9

[42] A. R. Cavalli, B.-M. Chin, and K. Chon. Testing methods for sdl systems. *Computer Networks and ISDN Systems*, 28(12):1669–1683, 1996. ISSN 0169-7552. doi: 10.1016/0169-7552(95)00125-5. URL `http://dx.doi.org/10.1016/0169-7552(95)00125-5`. 9

[43] J.-C. Fernandez, C. Jard, T. Jéron, and C. Viho. Using on-the-fly verification techniques for the generation of test suites. In *Proceedings of the 8th International Conference on Computer Aided Verification*, CAV '96, pages 348–359, London, UK, UK, 1996. Springer-Verlag. ISBN 3-540-61474-5. URL `http://dl.acm.org/citation.cfm?id=647765.735847`. 9

[44] J. Tretmans. Testing concurrent systems: A formal approach. In JosC.M. Baeten and Sjouke Mauw, editors, *Concurrency Theory*, Lecture Notes in Computer Science - 1664, pages 46–65. Springer Berlin Heidelberg, 1999. ISBN 978-3-540-66425-3. doi: 10.1007/3-540-48320-9_6. URL `http://dx.doi.org/10.1007/3-540-48320-9_6`. 9

[45] B. Nielsen and A. Skou. Automated test generation from timed automata. In *Proceedings of the 7th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS '01, pages 343–357, London, UK, UK, 2001. Springer-Verlag. ISBN 3-540-41865-2. URL `http://dl.acm.org/citation.cfm?id=646485.694454`. 10

[46] M. Krichen and S. Tripakis. Conformance testing for real-time systems. *Formal Methods System Design*, 34(3):238–304, 2009. ISSN 0925-9856. doi: 10.1007/s10703-009-0065-1. URL `http://dx.doi.org/10.1007/s10703-009-0065-1`. 10

[47] R. Castanet, O. Kone, and P. Laurencot. On the fly test generation for real time protocols. In *Proceedings of the 1998 7th International Conference on Computer Communications and Networks*, pages 378 –385, October 1998. doi: 10.1109/ICCCN.1998.998798. 10

[48] T. Jéron. Symbolic model-based test selection. *Electronic Notes in Theoretical Computer Science*, 240:167–184, 2009. ISSN 1571-0661. doi: 10.1016/j.entcs.2009.05.051. URL `http://dx.doi.org/10.1016/j.entcs.2009.05.051`. 10

[49] M.-C. Gaudel. Testing can be formal, too. In Peter D. Mosses, Mogens Nielsen, and Michael I. Schwartzbach, editors, *TAPSOFT 95: Theory and Practice of Software Development, Proceedings of the 6th International Joint Conference CAAP/FASE, Aarhus, Denmark, May 22-26, 1995*, Lecture Notes in Computer Science - 915, pages 82–96. Springer, May 1995. ISBN 3-540-59293-8. 10

[50] B. Marre. Loft: A tool for assisting selection of test data sets from algebraic specifications. In *Proceedings of the 6th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, TAPSOFT '95, pages 799–800, London, UK, UK, May 1995. Springer-Verlag. ISBN 3-540-59293-8. URL `http://dl.acm.org/citation.cfm?id=646619.697561`. 10

[51] B. Marre and A. Arnould. Test sequences generation from lustre descriptions: Gatel. In *Proceedings of the 2000 IEEE 15th International Conference on Automated Software Engineering*, ASE '00, pages 229–, Washington, DC, USA, 2000. IEEE Computer Society. ISBN 0-7695-0710-7. URL `http://dl.acm.org/citation.cfm?id=786768.786972`. 10

[52] G. Lestiennes and M.-C. Gaudel. Testing processes from formal specifications with inputs, outputs and data types. In *Proceedings of the 13th International Symposium on Software*

*Reliability Engineering*, ISSRE '02, pages 3–, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-8186-1763-3. URL http://dl.acm.org/citation.cfm?id=851033.856301. 10

[53] A. D. Brucker and B. Wolff. On theorem prover-based testing. *Formal Aspects of Computing*, 2012. ISSN 0934-5043. doi: 10.1007/s00165-012-0222-y. 10

[54] J. D. Musa. Operational profiles in software-reliability engineering. *IEEE Software*, 10(2): 14–32, 1993. ISSN 0740-7459. doi: 10.1109/52.199724. URL http://dx.doi.org/10.1109/52.199724. 10

[55] C. Wohlin and P. Runeson. Certification of software components. *IEEE Transactions on Software Engineering*, 20(6):494–499, 1994. ISSN 0098-5589. doi: 10.1109/32.295896. URL http://dx.doi.org/10.1109/32.295896. 10

[56] J. A. Whittaker and J. H. Poore. Markov analysis of software specifications. *ACM Transactions on Software Engineering Methodologies*, 2(1):93–106, 1993. ISSN 1049-331X. doi: 10.1145/151299.151326. URL http://doi.acm.org/10.1145/151299.151326. 10

[57] Matelo - product — all4tec. URL http://www.all4tec.net/index.php/All4tec/matelo-product.html. 10

[58] W. Dulz and F. Zhen. Matelo - statistical usage testing by annotated sequence diagrams, markov chains and ttcn-3. In *Proceedings of the 3rd International Conference on Quality Software*, QSIC '03, pages 336–, Washington, DC, USA, 2003. IEEE Computer Society. ISBN 0-7695-2015-4. URL http://dl.acm.org/citation.cfm?id=950789.951283. 10

[59] M. Beyer, W. Dulz, and Fenhua Zhen. Automated ttcn-3 test case generation by means of uml sequence diagrams and markov chains. In *12th Asian Test Symposium*, ATS '03, pages 102–105, November 2003. doi: 10.1109/ATS.2003.1250791. 10

[60] G.H. Walton, R.M. Patton, and D.J. Parsons. Usage testing of military simulation systems. In *Proceedings of the Winter Simulation Conference*, volume 1, pages 771–779, 2001. doi: 10.1109/WSC.2001.977366. 10

[61] A. Ost and D. van Logchem. Statistical usage testing applied to mobile network verification. In *2001 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '01, pages 160–163, 2001. doi: 10.1109/ISPASS.2001.990694. 10

[62] C. Kallepalli and J. Tian. Measuring and modeling usage and reliability for statistical web testing. *IEEE Transactions on Software Engineering*, 27(11):1023–1036, November 2001. ISSN 0098-5589. doi: 10.1109/32.965342. 10

[63] M. Gittens, H. Lutfiyya, and M. Bauer. An extended operational profile model. In *Software Reliability Engineering, 2004 15th International Symposium on*, ISSRE '04. 10

[64] Y. Falcone, J.-C. Fernandez, T. Jéron, H. Marchand, and L. Mounier. More testable properties. In *Proceedings of the IFIP WG 6.1 22nd International Conference on Testing Software and Systems*, ICTSS '10, pages 30–46, Berlin, Heidelberg, 2010. Springer-Verlag. ISBN 3-642-16572-9, 978-3-642-16572-6. URL http://dl.acm.org/citation.cfm?id=1928028.1928032. 11

[65] G. Durrieu, H. Waeselynck, and V. Wiels. Leto - a lustre-based test oracle for airbus critical systems. In D. D. Cofer and A. Fantechi, editors, *FMICS*, Lecture Notes in Computer Science - 5596, pages 7–22. Springer, 2008. ISBN 978-3-642-03239-4. URL http://dblp.uni-trier.de/db/conf/fmics/fmics2008.html#DurrieuWW08. 11, 125

[66] L. C. Briand, Y. Labiche, and H. Sun. Investigating the use of analysis contracts to support fault isolation in object oriented code. *SIGSOFT Software Engineering Notes*, 27(4): 70–80, 2002. ISSN 0163-5948. doi: 10.1145/566171.566183. URL http://doi.acm.org/10.1145/566171.566183. 11

[67] L. Du Bousquet, Y. Ledru, O. Maury, C. Oriat, J. l. Lanet, L. Bousquet, Y. Ledru, O. Maury, and C. Oriat. Case study in jml-based software validation. In *Proceedings of the 19th International Conference on Automated Software Engineering*. Press, 2004. 11

[68] Y. Le Traon, B. Baudry, and J.-M. Jezequel. Design by contract to improve software vigilance. *IEEE Transactions on Software Engineering*, 32(8):571–586, 2006. ISSN 0098-5589. doi: 10.1109/TSE.2006.79. URL http://dx.doi.org/10.1109/TSE.2006.79. 11

[69] M. Fewster and D. Graham. *Software Test Automation: Effective Use of Test Execution Tools*. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1999. ISBN 0-201-33140-3. 11, 12

[70] Test automation - silk test. URL http://www.borland.com/products/silktest/. 12

[71] Abbot framework for automated testing of java gui components and programs. URL http://abbot.sourceforge.net/doc/overview.shtml. 13

[72] A. Ruiz and Y. W. Price. Test-driven gui development with testng and abbot. *IEEE Software*, 24(3):51–57, May 2007. ISSN 0740-7459. doi: 10.1109/MS.2007.92. URL http://dx.doi.org/10.1109/MS.2007.92. 13

[73] Extensible markup language (xml) 1.0 (fifth edition), 2008. URL http://www.w3.org/TR/2008/REC-xml-20081126/. 13, 38

[74] Selenium - web browser automation. URL http://seleniumhq.org/. 13

[75] A. Sirotkin. Web application testing with selenium. *Linux Journal*, 2010(192), 2010. ISSN 1075-3583. URL http://dl.acm.org/citation.cfm?id=1767726.1767729. 13

[76] A. Holmes and M. Kellogg. Automating functional tests using selenium. In *Proceedings of the 2006 Conference on Agile Software Development*, AGILE '06, pages 270–275, Washington, DC, USA, 2006. IEEE Computer Society. ISBN 0-7695-2562-8. doi: 10.1109/AGILE.2006.19. URL http://dx.doi.org/10.1109/AGILE.2006.19. 13

[77] D. Xu, W. Xu, B. K. Bavikati, and W. E. Wong. Mining executable specifications of web applications from selenium ide tests. In *Proceedings of the 2012 IEEE 6$^{th}$ International Conference on Software Security and Reliability*, SERE '12, pages 263–272. IEEE, 2012. ISBN 978-0-7695-4742-8. URL http://dblp.uni-trier.de/db/conf/ssiri/sere2012.html#XuXBW12. 13

[78] Cunit home. URL http://cunit.sourceforge.net/. 13

[79] Sourceforge.net: cppunit. URL http://cppunit.sourceforge.net/. 14

[80] Junit. URL http://junit.sourceforge.net/. 14

[81] Pyunit - the standard unit testing framework for python. URL http://pyunit.sourceforge.net/. 14

[82] *Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language.* 14, 27, 30, 57, 135

[83] C. Willcock, T. Deiss, and S. Tobies. *An Introduction to TTCN-3.* Wiley, Chichester, 2005. 14, 15, 30, 73

[84] J. Grossmann, D. Serbanescu, and I. Schieferdecker. Testing embedded real time systems with ttcn-3. In *Proceedings of the 2009 International Conference on Software Testing Verification and Validation*, ICST '09, pages 81–90, Washington, DC, USA, 2009. IEEE Computer Society. ISBN 978-0-7695-3601-9. doi: 10.1109/ICST.2009.37. URL http://dx.doi.org/10.1109/ICST.2009.37. 14, 51

[85] Z. R. Dai, J. Grabowski, and H. Neukirchen. TimedTTCN-3 - A Real-Time Extension for TTCN-3. Technical report, SIIM Technical Report SIIM-TR-A-01-14, Schriftenreihe der Institute für Informatik/Mathematik, Medical University of Lübeck, Germany, 23. November 2001, 2001. 14

[86] I. Schieferdecker, E. Bringmann, and J. Gro. Continuous ttcn-3: Testing of embedded control systems. In *Proceedings of the 2006 International Workshop on Software Engineering for Automotive Systems*, SEAS '06, pages 29–36, New York, NY, USA, 2006. ACM. ISBN 1-59593-402-2. doi: 10.1145/1138474.1138481. URL http://doi.acm.org/10.1145/1138474.1138481. 14, 50, 51

[87] S. Schulz and T. Vassiliou-Gioles. Implementation of ttcn-3 test systems using the tri. In *Proceedings of the IFIP 14$^{th}$ International Conference on Testing Communicating Systems*, TestCom '02, pages 425–442, Deventer, The Netherlands, The Netherlands, 2002. Kluwer, B.V. ISBN 0-7923-7695-1. URL http://dl.acm.org/citation.cfm?id=648132.748152. 14, 33, 41

[88] Introduction to asn.1. URL http://www.itu.int/ITU-T/asn1/introduction/index.htm. 14

[89] Ibm - embedded software test automation framework - ibm rational test realtime - rational test realtime - software. URL http://www-01.ibm.com/software/awdtools/test/realtime/. 14

[90] M. Balcer, W. Hasling, and T. Ostrand. Automatic generation of test scripts from formal test specifications. *SIGSOFT Software Engineering Notes*, 14(8):210–218, 1989. ISSN 0163-5948. doi: 10.1145/75309.75332. URL http://doi.acm.org/10.1145/75309.75332. 14

[91] M. Bennett, R. Borgen, K. Havelund, M. Ingham, and D. Wagner. Development of a prototype domain-specific language for monitor and control systems. In *2008 IEEE Aerospace Conference*, pages 1–18, March 2008. doi: 10.1109/AERO.2008.4526660. 15

[92] T. Stahl, M. Voelter, and K. Czarnecki. *Model-Driven Software Development: Technology,*

*Engineering, Management.* John Wiley & Sons, 2006. ISBN 0470025700. 15, 28, 57

[93] Omg unified modeling language$^{TM}$ (omg uml) - infrastructure, version 2.4.1, January 2012. URL http://www.omg.org/spec/UML/2.4.1/. 15, 60

[94] Uml testing profile (utp), version 1.1. URL http://www.omg.org/spec/UTP/1.1/PDF. 15, 65

[95] J. Zander, Z. R. Dai, I. Schieferdecker, and G. Din. From u2tp models to executable tests with ttcn-3 - an approach to model driven testing. In *Proceedings of the 17th IFIP TC6/WG 6.1 International Conference on Testing of Communicating Systems*, TestCom '05, pages 289–303, Berlin, Heidelberg, 2005. Springer-Verlag. ISBN 3-540-26054-4, 978-3-540-26054-7. doi: 10.1007/11430230_20. URL http://dx.doi.org/10.1007/11430230_20. 15

[96] I. Schieferdecker and G. Din. A meta-model for ttcn-3. In Manuel Nez, Zakaria Maamar, Fernando L. Pelayo, Key Pousttchi, and Fernando Rubio, editors, *FORTE Workshops*, Lecture Notes in Computer Science - 3236, pages 366–379. Springer, 2004. ISBN 3-540-23169-2. URL http://dblp.uni-trier.de/db/conf/forte/fortew2004.html#SchieferdeckerD04. 15

[97] Ttworkbench - the reliable test automation platform, testing technologies. URL http://www.testingtech.com/products/ttworkbench.php. 15

[98] P. Baker and C. Jervis. Testing uml2.0 models using ttcn-3 and the uml2.0 testing profile. In *Proceedings of the 13th International SDL Forum Conference on Design for Dependable Systems*, SDL '07, pages 86–100, Berlin, Heidelberg, 2007. Springer-Verlag. ISBN 3-540-74983-7, 978-3-540-74983-7. URL http://dl.acm.org/citation.cfm?id=1779934.1779942. 15

[99] Rational tau, ibm. URL http://www01.ibm.com/software/awdtools/tau/. 15

[100] T. Ostrand, A. Anodide, H. Foster, and T. Goradia. A visual test development environment for gui systems. In *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '98, pages 82–92, New York, NY, USA, 1998. ACM. ISBN 0-89791-971-8. doi: 10.1145/271771.271793. URL http://doi.acm.org/10.1145/271771.271793. 16, 50

[101] Y. Yongfeng, L. Bin, Z. Deming, and J. Tongmin. On modeling approach for embedded real-time software simulation testing. *Systems Engineering and Electronics, Journal of*, 20(2):420–426, April 2009. 16

[102] C. Efkemann and J. Peleska. Model-based testing for the second generation of integrated modular avionics. In *Proceedings of the 2011 IEEE 4th International Conference on Software Testing, Verification and Validation Workshops*, ICSTW '11, pages 55–62, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4345-1. doi: 10.1109/ICSTW.2011.72. URL http://dx.doi.org/10.1109/ICSTW.2011.72. 16

[103] Scarlett project, . URL hhttp://www.scarlettproject.eu. 16

[104] Rt-tester 6.x product information. URL http://www.verified.de/_media/en/products/rt-tester_information.pdf. 16

[105] M. Dahlweid, O. Meyer, and J. Peleska. Automated testing with rt-tester - theoretical issues driven by practical needs. In *In Proceedings of the FM-Tools 2000, number 2000-07 in Ulmer Informatik Bericht*, 2000. 16

[106] J. Fischer, M. Piefel, and M. Scheidgen. A metamodel for sdl-2000 in the context of metamodelling ulf. In Daniel Amyot and Alan W. Williams, editors, *Proceedings of the 4th International SDL and MSC Conference on System Analysis and Modeling*, Lecture Notes in Computer Science - 3319, pages 208–223. Springer, 2004. ISBN 3-540-24561-8. URL http://dblp.uni-trier.de/db/conf/sam/sam2004.html#FischerPS04. 16

[107] J. Großmann, P. Makedonski, H.-W. Wiesbrock, J. Svacina, I. Schieferdecker, and J. Grabowski. *Model-Based X-in-the-Loop Testing*, chapter 12. CRC Press, 2011. 16

[108] A. Ott. *System Testing in the Avionics Domain*. Doctoral dissertation, Universität Bremen, 2007. Available at http://nbn-resolving.de/urn:nbn:de:gbv:46-diss000108814. 19

[109] I. Land and J. Elliott. *Architecting ARINC 664, Part 7 (AFDX) Solutions.* Xilinx, May 2009. 22

[110] Arinc specification 429p1-18 digital information transfer system (dits), part 1, functional description, electrical interfaces, label assignments and word formats. URL https://www.arinc.com/cf/store/catalog_detail.cfm?item_id=1941. 22

[111] Mil-std-1553 digital time division command/response multiplex data bus. URL `http://quicksearch.dla.mil/basic_profile.cfm?ident_number=36973&method=basic`. 22

[112] Ieee standard c/atlas test language. *IEEE Std 716-1989*, 1989. doi: 10.1109/IEEESTD.1989.101066. 27, 57, 135

[113] Ieee standard for automatic test markup language (atml) for exchanging automatic test equipment and test information via xml. *IEEE Std 1671-2010 (Revision of IEEE Std 1671-2006)*, pages 1–388, 20 2011. doi: 10.1109/IEEESTD.2011.5706290. 27, 57, 135

[114] J. Grossmann, I. Fey, A. Krupp, M. Conrad, C. Wewetzer, and W. Mueller. Model-driven development of reliable automotive services. chapter TestML - A Test Exchange Language for Model-Based Testing of Embedded Software, pages 98–117. Springer-Verlag, Berlin, Heidelberg, 2008. ISBN 978-3-540-70929-9. doi: 10.1007/978-3-540-70930-5_7. URL `http://dx.doi.org/10.1007/978-3-540-70930-5_7`. 30, 135

[115] Xml schema 1.1. URL `http://www.w3.org/XML/Schema.html`. 33

[116] Mathworks france - matlab - le langage du calcul scientifique. URL `http://www.mathworks.fr/products/matlab/`. 33

[117] Y. Hernandez, T. M. King, J. Pava, and P. J. Clarke. A meta-model to support regression testing of web applications. In *Proceedings of the 20th International Conference on Software Engineering Knowledge Engineering, San Francisco, CA, USA, July 1-3, 2008*, SEKE '08, pages 500–505. Knowledge Systems Institute Graduate School, 2008. ISBN 1-891706-22-5. 50

[118] A. Guduvan, H. Waeselynck, V. Wiels, G. Durrieu, M. Schieber, and Y. Fusero. A meta-model for tests of avionics embedded systems. *to appear in Proceedings of MODELSWARD 2013 1st International Conference on Model-Driven Engineering and Software Development*, February 2013. 59

[119] Eclipse modeling emft - home. URL `http://www.eclipse.org/modeling/emft/?project=ecoretools`. 60, 85, 93

[120] Omg mof 2 xmi mapping specification, version 2.4.1, August 2011. URL `http://www.omg.org/spec/XMI/2.4.1/`. 60

[121] Graphical modeling framework. URL `http://www.eclipse.org/modeling/gmp/`. 60, 86, 92

[122] Graphiti home. URL `http://www.eclipse.org/graphiti/`. 60, 86, 92

[123] Xtext. URL `http://www.eclipse.org/Xtext/`. 60, 82, 85, 86

[124] Object constraint language, version 2.3.1, January 2011. URL `http://www.omg.org/spec/OCL/2.3.1/`. 60, 91

[125] Acceleo. URL `http://www.eclipse.org/acceleo/`. 60, 93, 119

[126] Xpand - eclipse. URL `http://www.eclipse.org/modeling/m2t/?project=xpand/`. 60

[127] R. Johnson and B. Woolf. The Type Object Pattern, 1997. 61, 91, 107

[128] Atl. URL `http://www.eclipse.org/atl/`. 93

[129] Object management group. URL `http://www.omg.org/`. 94

[130] Omg's metaobject facility (mof) home page. URL `http://www.omg.org/mof/`. 94

[131] Mof model to text transformation language (mofm2t), version 1.0, January 2008. URL `http://www.omg.org/spec/MOFM2T/1.0/`. 94

[132] K. Czarnecki and S. Helsen. Feature-based survey of model transformation approaches. *IBM Systemts Journal*, 45(3):621–645, 2006. ISSN 0018-8670. doi: 10.1147/sj.453.0621. URL `http://dx.doi.org/10.1147/sj.453.0621`. 98, 119, 120

[133] S. Zschaler and A. Rashid. Towards modular code generators using symmetric language-aware aspects. In *Proceedings of the 1st International Workshop on Free Composition*, FREECO '11, pages 6:1–6:5, New York, NY, USA, 2011. ACM. ISBN 978-1-4503-0892-2. doi: 10.1145/2068776.2068782. URL `http://doi.acm.org/10.1145/2068776.2068782`. 99, 120

[134] Scade suite :: Products, . URL `http://www.esterel-technologies.com/products/scade-suite/`. 120

[135] A. Guduvan, H. Waeselynck, V. Wiels, G. Durrieu, M. Schieber, and Y. Fusero. Stelae - a model-driven test development environment for avionics systems. *to appear in Proceedings of ISORC 2013 : 16th IEEE Computer Society International Symposium on Object Component Service-oriented Real-time Distributed Computing*, June 2013. 123

## Abstract

The development of tests for avionics systems involves a multiplicity of in-house test languages, with no standard emerging. Test solution providers have to accommodate the habits of different clients, while the exchange of tests between aircraft manufacturers and their equipment/system providers is hindered. We propose a model-driven approach to tackle these problems: test models would be developed and maintained in place of code, with model-to-code transformations towards target test languages. This thesis presents three contributions in this direction. The first one consists in the analysis of four proprietary test languages currently deployed. It allowed us to identify the domain-specific concepts, best practices, as well as pitfalls to avoid. The second contribution is the definition of a meta-model in EMF Ecore that integrates all identified concepts and their relations. The meta-model is the basis for building test model editors and code generation templates. Our third contribution is a demonstrator of how these technologies are used for test development. It includes customizable graphical and textual editors for test models, together with template-based transformations towards a test language executable on top of a real test platform.