STELAE - A Model-Driven Test Development Environment for Avionics Systems

Alexandru Robert Ciprian Guduvan, Hélène Waeselynck, V. Wiels, Gaël Durrieu, Yann Fusero, Michel Schieber

STELAE – A Model-Driven Test Development Environment for Avionics Systems

Alexandru-Robert GUDUVAN[1,2,3], Hélène WAESELYNCK[2,3], Virginie WIELS[4], Guy DURRIEU[4], Yann FUSERO[1] and Michel SCHIEBER[1]

[1]*Cassidian Test & Services, 5 av. de Guynemer, B.P. 86, F-31772 Colomiers Cedex, France*
[2]*CNRS, LAAS, 7 av. du Colonel Roche, F-31400 Toulouse, France*
[3]*Univ de Toulouse, LAAS, F-31400 Toulouse, France*
[4]*ONERA/DTIM, 2 av. Edouard Belin, B.P. 74025, F-31055 Toulouse Cedex 4, France*
Alexandru-Robert.Guduvan@cassidian.com, Helene.Waeselynck@laas.fr, Virginie.Wiels@onera.fr, Guy.Durrieu@onera.fr, Yann.Fusero@cassidian.com, Michel.Schieber@cassidian.com

***Abstract*—In this paper we present STELAE, a model-driven test development environment for avionics embedded systems, implemented on top of a real integration test platform. It is the result of an R&D project between two research laboratories and a test solution provider, aiming to introduce model-driven engineering methodologies and technologies for the development of tests. Our work was motivated by the multiplicity of proprietary test languages in this industrial context, which no longer respond to the stakeholder needs. We present the early prototype functionalities (test model definition, automatic code generation and execution) on a case study inspired from real-life. Our feedback on the used technologies concludes this paper.**

Keywords—test development, test language, test model, model-driven engineering, development environment, automatic code generation

I. INTRODUCTION

This work deals with the implementation of tests for avionics embedded systems. The current practice is heterogeneous, as it involves a multiplicity of in-house test languages to code the tests. Test solution providers, equipment/system providers and aircraft manufacturers all have their own proprietary test languages and associated tools. No standardized test language has emerged, in contrast to other fields that use international standards, for example: the Abbreviated Test Language for All Systems (ATLAS) [1] and the Automatic Test Markup Language (ATML) [2] standards in hardware testing or the Testing and Test Control Notation Version 3 (TTCN-3) [3] in the field of telecommunication protocols and distributed systems. These standards are not designed to address the specificities of our industrial context and as such are not directly reusable. The multiplicity of proprietary test languages is challenging for the different stakeholders of the avionics industry. Test solution providers have to accommodate the habits of different clients. The exchange of tests between aircraft manufacturers and equipment/system providers is hindered. A number of high-level needs (portability, maintainability and customizability) are not answered by existing solutions.

These issues have been the basis for launching a three-year R&D project involving a test solution provider and two research laboratories. The aim is to introduce a model-driven approach for test development, responding to this wide range of needs. Model-driven engineering is a means to abstract away from the existing proprietary implementation solutions. It promotes the central role of platform-independent models in the development activity: models are developed, maintained and shared. The proposed shift from test code to test models is driven by the fact that test software is indeed software, and that test development can benefit from advanced software engineering methodologies [4].

The approach is based on the definition of a meta-model that captures the domain-specific concepts and constrains the building of models, in the same way that a language grammar constrains the writing of code. The employed meta-modeling technology also gave us access to a wide range of free open-source tools that led to the rapid development of our prototype, called STELAE (Systems TEst LAnguage Environment).

STELAE gave us the opportunity to experiment with model-driven engineering and associated technologies. This first prototype allows us to demonstrate different test engineer activities:

- The **definition of test models** conforming to a test meta-model. We presented the test meta-model in [5]. We defined it using *Ecore* [6]. The test meta-model integrates a rich set of domain-specific concepts identified by our analysis of a set of proprietary test languages [7]. Test models are also analyzed according to a set of rules we defined in the *Object Constraint Language* (OCL) [8]. The definition of test models is performed:
    - In a graphical editor for structural test elements. The graphical editor was developed using the *Eclipse Modeling Framework* (EMF) [6].
    - In a textual editor for behavioral test elements. The textual editor was developed using *Xtext* [9].
- The **implementation of test models** through model-to-text transformations with template-based automatic

code generation. We used *Acceleo* [10] for the implementation. The target is a Python-based executable test language developed at Cassidian Test & Services.

- The **execution of the automatically generated files and code** on top of a real integration test platform: the U-TEST Real-Time System [11] developed by Cassidian Test & Services.

This paper presents the functionalities of STELAE on a case study inspired from real-life. It also gives our feedback on the open-source model-driven engineering technologies with which we experimented.

Section II introduces the industrial context. Section III presents a case study with two test cases that will guide the rest of the paper. Sections IV to VII discuss the functionalities offered by STELAE to test engineers, exemplified on the case study. Section VIII deals with related work. Section IX comprises our feedback on the different technologies that were employed and concludes this paper.

II. INDUSTRIAL CONTEXT

An avionics embedded system is typically a distributed system, with interconnected hardware elements: interconnected processors, memory modules, input/output cards, power supply devices, and others. Software elements running on the processors implement the functional logic. Among the verification and validation activities (for an overview see [12]) that accompany the system development process our focus is on the in-the-loop testing phases, which come in various forms: model/software/hardware-in-the-loop.

Avionics embedded systems have a predominantly reactive behavior: there are execution cycles to read the input data and compute the output ones. The system functionalities cannot be exercised unless all expected inputs are received from the environment at each cycle, with some time tolerance. This is the motivation for in-the-loop testing: the system under test (SUT) is coupled to a model of its environment that produces the data, together forming a (cyclic) closed-loop system.

As mentioned previously, standard test languages used in other fields are not directly reusable in ours. ATLAS and ATML target electronic circuitry manufacturing defects, detected by applying electrical signals at various places inside the circuit; while TTCN-3 targets the testing of open-loop systems, which are quiescent unless activated by some asynchronous messages.

In the avionics domain, communication between system components is achieved by buses, such as: AFDX (Avionics Full-Duplex Switched Ethernet) or ARINC 429 (Aeronautical Radio, Incorporated). The interfaces of a system are defined inside an Interface Control Document (ICD) (Figure 1). This document is organized into several hierarchical levels. Lower levels comprise connectors with pins. They are followed by buses attached to the pins. The higher levels comprise bus messages transporting application parameters as payload. As application parameters are meaningful to engineers, they are also called engineer variables. We shall refer to them as such in the rest of the paper. ICD elements are distinguished by unique string identifiers built from a path name traversing the tree-like structure of the ICD. Such identifiers provide an abstraction for accessing the SUT interfaces. For an engineer variable, an identifier would have the generic form:

`'SUT/BUS/MESSAGE/ENG_VARIABLE'.`

Various predefined test actions can be applied to SUT interfaces, at all of the ICD hierarchical levels (e.g. `set/getValue()` and timed stimulations, such as `ramp()`, for engineer variables; fault injection: `start/stopEmission()` on a bus).

In order to propose a model-driven approach for the development of tests for the in-the-loop testing of avionics embedded systems, we had to define the test meta-model underlying the approach. First we analyzed the current practice, by looking at the features offered by a sample of test languages currently deployed [7]. The domain-specific concepts issued from this analysis were afterwards integrated within the test meta-model [5]. In this paper we focus on the prototype test model development environment we implemented, which is called STELAE.

Interface Control Document (ICD)
SUT: ADIRS

#	CONNECTOR	BUS	LINE TYPE
CONNECTOR	CONNECTOR_1	ARINC_429_IN_1	ARINC_429

#	BUS	BUS CONFIGURATION	CONNECTOR
ARINC 429 INPUT BUS	ARINC_429_IN_1	…	CONNECTOR_1

#	MESSAGE	BUS	ENGINEER VARIABLE
ARINC 429 INPUT LABEL	LABEL_IN_1	ARINC_429_IN_1	AC_SPEED_1

Figure 1. ADIRS-Inspired ICD Example Snippet

III. CASE STUDY

For our demonstration of the STELAE functionalities, we chose a case study inspired from a real one targeting the ADIRS (Air Data Inertial Reference System) [13]. We developed a simulation of part of the ADIRS behavior. We implemented this case study on a real integration test platform: U-TEST Real-Time System [11]. 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 also tested our approach on a real case study targeting the Flight Warning System, where we verify the synthesis of a global alarm from partial alarms in an aircraft engine fire situation. We do not present it in this paper as it would have allowed us to show only a limited number of concepts in comparison with the ADIRS case study.

ADIRS deals with the acquisition of several engineer variables necessary for the flight control system (e.g., altitude,

speed, angle of attack). For each of these engineer variables redundant sensors exist and a consolidated value is computed from the set of available input values.

We deal here with the aircraft speed engineer variable. The values of three input engineer variables (`AC_SPEED_1/2/3`) are used to compute the value of the output consolidated engineer variable (`AC_SPEED`). The ADIRS logic is the following:

- Nominal behavior: 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.
- Degraded behavior: 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 behavior:

- **Nominal behavior 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 behavior 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 engineer variables.

For exemplification purposes, we assume that our simplified ADIRS employs ARINC 429 buses for transporting its input and output engineer variables.

IV. Test Solution Provider and User Separation

One of the main architectural choices we made was to separate the elements related to the test solution provider from those related to the test solution user inside the test meta-model. A high-level view of the test meta-model structure can be found in Figure 2. This separation allows the test solution provider to easily customize and maintain the test solution, by rendering available to test solution users different SUT interface types (e.g., types of buses, messages, engineer variables) and associated test actions (e.g. set and get the value of an engineer variable, start and stop the emission on a bus).

For our case study, let us assume that the following elements are already available to the test engineer: the AFDX bus type and the float engineer variables with the following predefined test actions: `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 engineer variables. Consequently, the test solution provider can add them to the list of already existing ones using our predefined extension points. Test actions that do not correspond to interactions with the SUT are distributed inside toolkit structures. Such an example is the `waitDuration()` functionality, attached to a time management toolkit, that the test solution provider also makes available. Figure 3 shows the functionalities that are rendered available by the test solution provider for the example discussed above. In STELAE, a password-based access control system restricts access to the test solution provider section.

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.

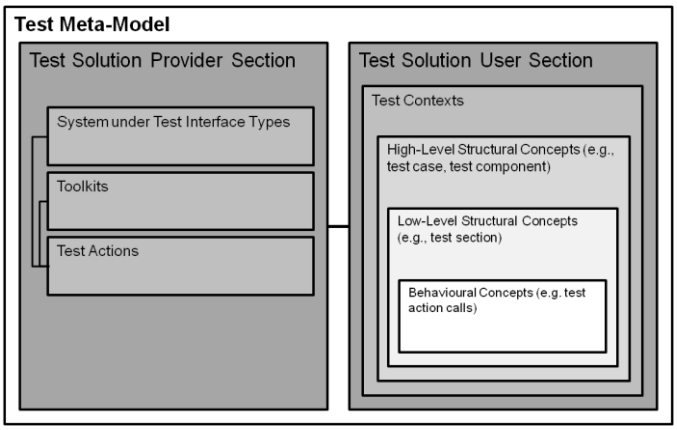

Figure 2. Test Meta-Model High-Level View

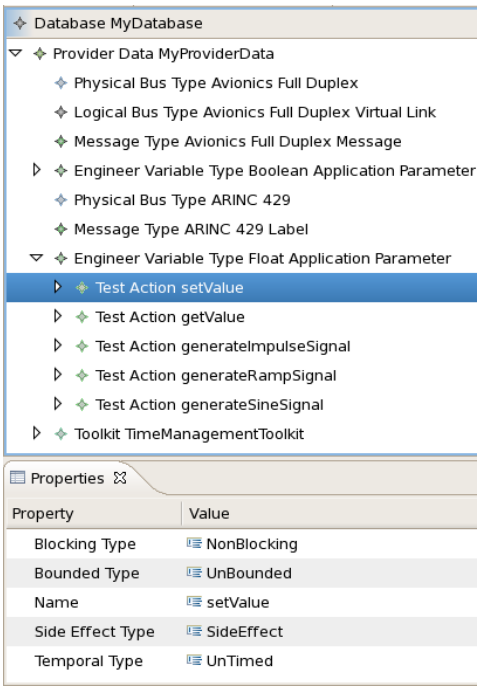

Figure 3. Test Solution Provider - SUT Interface Types and Test Actions

V. TEST STRUCTURE MODELING

First we present the different concepts that a test engineer has access to in STELAE, and afterwards we show how these concepts are actually used for the case study modeling.

For the definition of the structural aspects, the test engineer employs the graphical editor (Figure 8.a). For our prototype this editor is basic. It comprises a tree-like navigable view of test structural elements, with contextual menus and input data fields. In order to begin modeling the two test cases of our case study, the test engineer must first define a test context. A test context is a container for test cases applied to a SUT, together with an architecture of test components. This concept is inspired from the UML Testing Profile [14]. A conceptual view can be found in Figure 4. We call the elements of a test context high-level structural elements.

Test components are executable elements that run in parallel during a test. They can access the interfaces of the SUT and call test actions. In addition, they also have access to an external pool of events (i.e., for synchronization) and shared data (i.e., for communication). For fault avoidance purposes, we defined a one producer – many consumers policy. Clashes can thus be detected by means of OCL rules, for example when several test components target a same SUT interface with test actions that have side effects. Notice the `SideEffectType` attribute in the "Properties" view in Figure 3.

A test component can be instantiated several times inside a test case, with the test case being in control of the execution of each test component instance. Only the test case is able to start, stop or pause the execution of a test component instance.

For ever higher reuse capabilities, test components possess formal interfaces that we call accessors. It is the test architecture associated to a test case that indicates the connection between the formal interfaces and the interfaces of the SUT or the pool of events and shared data.

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`. Each test component is instantiated once within each previously mentioned test cases: `Nominal_Component_1` and respectively `Degraded_Component_1`.

In order to render the test components reusable, we add four formal interfaces to each one: three for the first input engineer variables (`First_IN`, `Second_IN` and `Third_IN`) and one for the output engineer variable (`OUT`). The connection to the corresponding permutations of input and output engineer variables of the ADIRS is defined within the test architectures owned by the test cases. Figure 4 shows the corresponding conceptual view for the nominal test case.

VI. TEST BEHAVIOR MODELING

Two types of behavior can be modeled: for the test case and for the test components.

Test cases are in charge of controlling the execution of test component instances. In our simplified case study the test cases execute a `startExecutableElement()` command on the two test component instances.

Let us now look at our two test components. First it is important to mention that our analysis of test languages revealed the fact that test engineers are accustomed to using high-level predefined test component constructs that hide the low-level multi-threading aspects. We identified three types of test components: simple ones such as sequential test components and test monitors, as well as the timed periodic test component type. A periodic test component executes the same behavior periodically, while the test monitor has a simple behavior of the form `condition-> action`. Following discussions with test engineers, we also defined a new test component type: the cycle-by-cycle test component.

The test meta-model integrates all of these test component types, although we illustrate only two of them in this paper. The `Nominal_Component` is a sequential test component, while the `Degraded_Component` is a cycle-by-cycle test component. The sequential test component executes its behavior only once, while the cycle-by-cycle test component has different behaviors for each SUT execution cycle or set of cycles. It can be "synchronized" with the SUT execution cycles.

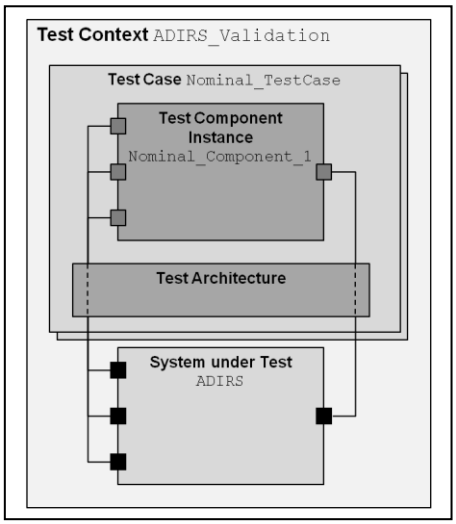

Figure 4. Conceptual View of the Test Context

Each test component type has its behavior organized inside low-level structural elements. A sequential test component organizes its behavior within sequential blocks. The sequential blocks are executed one after the other, with each sequential block comprising a list of statements. Each sequential block can correspond to a different phase performed during a test, such as SUT initialization and stimulation. We defined three sequential blocks for the `Nominal_Component` sequential test component: `Initialization`, `Stimulation` and `Behavior`. The first 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 second

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 last sequential block verifies that the value for the output parameter (`OUT`) is the median.

A cycle-by-cycle test component comprises elements that allow test engineers to precisely define the behavior of the test component "synchronized" with each cycle of the SUT, such as: cycle, repeated cycle or iterated cycle. A cycle has a behavior to be executed only once, for one of the SUT cycles. A repeated cycle has a behavior to be executed several times, depending on the evaluation of a logical condition. For fault avoidance purposes, we constrain the repeated cycle to be bound by a maximum number of times it is executed. An iterated cycle has a behavior to be executed for a fixed number of times. Figure 5 exemplifies the cycle-by-cycle behavior 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 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. Figure 6 shows the corresponding model in the STELAE graphical editor, while Figure 7 shows the behavior of the `Initialization` cycle in the STELAE textual editor. It is important to mention that the concrete syntax found in Figure 7 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.

It is worthy to mention that special instructions are used for verifications of the SUT behavior inside a test component (e.g., for verifying that the output value is the median). The results of these verifications lead to the definition of a test verdict. A test verdict can have one of the five following possible values, among which an order relation was defined: `none` > `pass` > `inconclusive` > `fail` > `error`. This relation allows the automatic synthesis of a global verdict from local ones: the verdict of a test case is computed from the verdict of the different test component instances it possesses. For this verdict management we took inspiration from TTCN-3 [3]. In the `Degraded_Component` we have two verifications. If one of these verifications leads to a `pass` and the other to a `fail` then the local verdict of the test component is `fail`. As our `Degraded_TestCase` only has one test component instance, then its global verdict would be `fail` as well.

Cycle	#0	#1	#2 to #4	#5
Behavior	First_IN = 10 Second_IN = 25 Third_IN = 34	Second_IN = 40	Nothing	Verify (OUT == 22)

Cycle	#6	#7 to #9	#10
Behavior	Second_IN = 25	Nothing	Verify (OUT != 25)

Figure 5. `Degraded_Component` Behavior Description

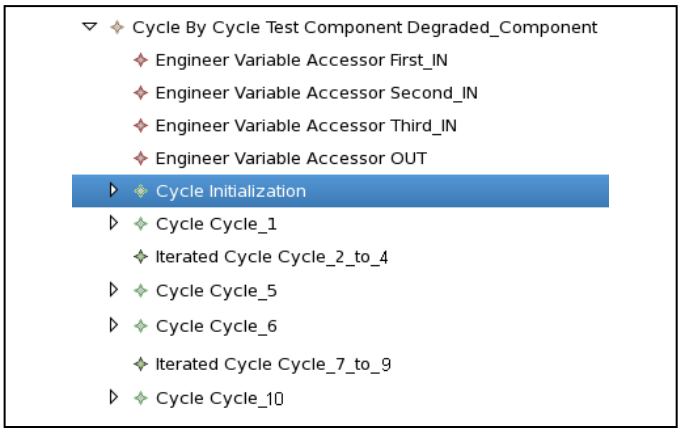

Figure 6. `Degraded_Component` in Graphical Editor

```
CallAccessor First_IN.setValue( 10 )
CallAccessor Second_IN.setValue ( 25 )
CallAccessor Third_IN.setValue ( 34 )
```

Figure 7. `Initialization` Cycle Behavior in Textual Editor

We also implemented some fault avoidance functionalities in STELAE, in order to identify and help in the removal of test model problems, at design level. We previously gave an example concerning clash detection.

In addition, a partition for specifiable behavior was defined on the different low-level structural elements depending on the test component type to which they are attached. For example, loops are always bounded in the case of periodic and cycle-by-cycle test components, while this constraint is relaxed for sequential test components, where bounding is only optional.

For our periodic and cycle-by-cycle test components we also defined verifications that guarantee that the corresponding behavior is executed within the different periodicity time constraints. For example, we verify that the execution of the statements within a cycle does not exceed the duration of the cycle. If this happens, then the runtime automatically sets the verdict to `error`, informing the test engineer that the performance expected for the execution of the test was not met by the test platform. In addition to these verifications performed at runtime, we can also analyze the correctness of the test specification. For example, if a test action on a SUT interface, such as sine, is called within a cycle with a duration higher than the duration of the cycle, then it is clear that this specification is incorrect. This problem would be detected at runtime, but it is more useful to detect it before. In the case of the `Degraded_Component` such a rule is validated trivially, as we have no timed test actions that are being called. The OCL rules that we defined allowed us to analyze aspects such as those mentioned above.

VII. STELAE Environment Overview

Figure 8.a shows a screenshot of STELAE with the graphical ("UserData" view) and textual editors ("Behavior" view), on the `Nominal_TestCase` example. The "Console" view shows the execution traces of the automatically generated code for the `Degraded_TestCase`. Notice the two `pass`

local verdicts corresponding to the two verifications inside the `Degraded_Component`. The "Tests Management" view shows the two test cases with their corresponding global verdicts (both are `pass`). The automatically generated files are seen on the left ("Model Projects" view). As mentioned previously, the STELAE prototype was integrated into the real integration test platform U-TEST Real-Time System [11] developed by Cassidian Test & Services. STELAE was plugged in the Man-Machine Interface (MMI) software component of the test platform as an Eclipse perspective. The automatic code generation targeted a real Python-based test language executable on the test platform.

The graphical editor offers intelligent contextual menus and data input fields, while the textual editor offers syntax checking, auto-completion and code coloration functionalities.

As can be seen in Figure 8.a – the central tree-like editor, we implemented several other test cases for the ADIRS in order to test our approach. This figure also shows the test context for our second case study concerning the Flight Warning System (FWS).

For the `Nominal_TestCase`, Figure 8.b 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 engineer variables 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. Notice the sine timed stimulation on `AC_SPEED_2` in the first "Oscilloscope" view. The second "Oscilloscope" shows the values for the `AC_SPEED_1/2/3_STATUS` variables. The last "Oscilloscope" view shows the value for the `AC_SPEED_OUT` engineer variable. Notice its constant value, not influenced by the minor sine fluctuations on one of the input engineer variables.

VIII. Related Work

Model-driven engineering is an active field of research. We focus here on work addressing the use of model-driven engineering for the development and implementation of tests. Work addressing the selection of abstract tests from system models (model-based testing) is outside the scope.

Most existing work on test development solutions uses UML for the test models. Many projects have addressed the integration of the standardized UML Testing Profile [14] and TTCN-3 [3]. Among these projects, [15] uses the profile in order to produce TTCN-3 code (or code skeletons). In addition, a meta-model for TTCN-3 can be found in [16], later encapsulated within the TTworkbench platform [17]. A similar project at Motorola [18] uses the TAU tool suite [19].

Some authors proposed their own UML profiles. A UML profile and model transformations for web applications testing is discussed in [20]. In avionics, UML-based modeling of simulation software for model-in-the-loop testing is proposed in [21]. Also in avionics, [22] proposes test models conforming to a test meta-model (integrating automata-based formalisms), for the second generation of Integrated Modular Avionics (IMA). Previous work by the same authors includes automata-based test modeling for their RT-Tester platform [23] [24].

Neither UTP and UML, nor the various standardized test languages deployed in other domains do not offer the specific concepts our test engineers require. Moreover, UML does not seem to be a solution showcased by our industrial context. Consequently, the domain-specific concepts derived from our analysis of current practice were integrated inside our own test meta-model.

IX. Feedback on Used Technologies

The completion of STELAE required a total effort of 12 man-months. A team of five people, with four distinct roles, was involved in the project: the test meta-model architect, a software architect, a project manager and a developer. The distribution of the required effort for the different parts of our work was as follows:

- ~6 man-months (50%) for the definition of the test meta-model, for a person without any prior experience in meta-modeling/modeling (the list of domain-specific concepts was known at the beginning of this activity),
- ~1 man-month (9%) for the development of the automatic code generation template-based implementation,
- ~3 man-months (25%) for the development of the graphical and textual editors and their integration within the MMI component of the U-TEST Real-Time System, for a person with knowledge of the software architecture of the target test platform, but with minimal knowledge on the model-driven technologies that were used,
- ~½ man-months (4%) for software architecture definition,
- ~1 man-month (8%) for project management,
- ~½ man-months (4%) for other activities (test model verification rules definition, use-case implementation and STELAE testing).

Most of the effort concerned the definition of the test meta-model. One of the challenges was to homogenously integrate all of the domain-specific concepts we had previously identified. We achieved this objective, but the resulting test meta-model is complex. It currently contains 190 EClass elements representing the different concepts. 340 EAttribute and EReference elements formalize their different characteristics and relations. 18 EENum elements were included as well. The size of the test meta-model exceeds that of other meta-models or domain-specific languages discussed in literature. For future industrialization purposes this complexity could be avoided by developing wizards to guide the test engineer and automatically instantiate a skeleton test model.

The development of the graphical and textual editors was very fast, as this first prototype required only basic functionalities. In their current state, the editors do not yet offer test engineers all the functionalities/shortcuts they would need. Our evaluation of the development of richer, more ergonomic editors, with technologies such as Graphical Modeling Framework (GMF) or Graphiti [25], 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.

As mentioned previously, we used model to text transformations for the implementation of our test models. 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 (in our case the test cases of the case study), then define the expected target (what the corresponding files and code are) and only afterwards develop the templates that map the two [26]. Our experience confirms this, as we encountered no difficulty when developing the templates while being guided by the use case. Currently, 40% of the concepts present in the test meta-model have been implemented. The missing concepts where not implemented as the simple/medium complexity case study did not require them. Moreover, we targeted a relatively young 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 75 Acceleo modules, each with one template.

In conclusion, with limited previous experience with model-driven technologies, the STELAE project workgroup succeeded in the implementation of a first prototype. We can currently demonstrate the test model definition, automatic code generation and execution of simple to medium complexity test cases, on a real integration test platform.

ACKNOWLEDGEMENTS

The authors would like to thank all those implicated in the STELAE project: Gilles BALLANGER, Guilhem BONNAFOUS, Mathieu GARCIA and Etienne ALLOGO.

REFERENCES

[1] 716-1995 - IEEE Standard Test Language for All Systems - Common/Abbreviated Test Language for All Systems (C/ATLAS)

[2] 1671-2010 - IEEE Standard for Automatic Test Markup Language (ATML) for Exchanging Automatic Test Equipment and Test Information via XML

[3] ES 201 873 - Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language. 2012

[4] Thomas Stahl, Markus Voelter, and Krzysztof Czarnecki. 2006. Model-Driven Software Development: Technology, Engineering, Management. John Wiley & Sons.

[5] Guduvan, A., Waeselynck, H., Wiels, V., Durrieu, G., Schieber, M., and Fusero, Y.: 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, SciTePress, 9 pages, February 2013

[6] Eclipse Modeling - EMFT - Home, http://www.eclipse.org/modeling/emft/?project=ecoretools

[7] Guduvan, A., Waeselynck, H., Wiels, V., Durrieu, G., Schieber, M., and Fusero, Y.: Test Languages for In-the-Loop Testing of Avionic Embedded Systems, LAAS Report N°12151, Mars 2012, 21p. http://homepages.laas.fr/waeselyn/Reports/TR-12151.pdf

[8] OCL, Object Constraint Language, Version 2.3.1, January 2012, http://www.omg.org/spec/OCL/2.3.1/

[9] Xtext, http://www.eclipse.org/Xtext/

[10] Acceleo, http://www.eclipse.org/acceleo/

[11] Cassidian T & S - U-Test Software, http://www.eads-ts.com/web/products/software/utest.html

[12] Aliki Ott. System Testing in the Avionics Domain. Ph.D. Dissertation, University of Bremen, Germany, 2007

[13] Guy Durrieu, Hélène Waeselynck, Virginie Wiels. LETO - A Lustre-Based Test Oracle for Airbus Critical Systems. In Darren D. Cofer, Alessandro Fantechi, editors, Formal Methods for Industrial Critical Systems, 13th International Workshop, FMICS 2008, L Aquila, Italy, September 15-16, 2008, Revised Selected Papers. Volume 5596 of Lecture Notes in Computer Science, pages 7-22, Springer, 2008.

[14] UTP, UML Testing Profile, Version 1.1. 2012. http://www.omg.org/spec/UTP/1.1/

[15] J. Zander, Z. Ru Dai, I. Schieferdecker, G. Din. From U2TP models to executable tests with TTCN-3: An approach to model driven testing, in Proc. international conference on testing of communicating systems (TestCom 2005), pp. 289-303, 2005.

[16] Ina Schieferdecker, George Din. A Meta-model for TTCN-3. FORTE 2004 Workshops The FormEMC, EPEW, ITM, Toledo, Spain, October 1-2, 2004. Volume 3236 of Lecture Notes in Computer Science, pages 366-379, Springer, 2004

[17] TTworkbench - The Reliable Test Automation Platform, Testing Technologies. http://www.testingtech.com/products/ttworkbench.php

[18] Paul Baker and Clive Jervis, Testing UML2.0 Models Using TTCN-3 and the UML2.0 Testing Profile, Proc. SDL 2007, LNCS 4745, Springer, pp. 86-100, 2007.

[19] Rational Tau, IBM, http://www01.ibm.com/software/awdtools/tau/

[20] Yanelis Hernandez, Tariq M. King, Jairo Pava, Peter J. Clarke: A Meta-model to Support Regression Testing of Web Applications. SEKE 2008: 500-505

[21] Yin, Y. F., Liu, B., Zhong, D. M., & Jiang, T. M.. (2009). On modeling approach for embedded real-time software simulation testing. Journal of Systems Engineering and Electronics, 20(2), 420-426.

[22] 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.

[23] RT-Tester 6.X Product Information, URL:http://www.verified.de/_media/en/products/rt-tester_information.pdf.

[24] M. Dahlweid, O. Meyer, and J. Peleska. Automated Testing with RT-Tester – Theoretical Issus Driven by Practical Needs. In Proceedings of FM-Tools 2000, number 2000-07 in Ulmer Informatik Bericht, 2000.

[25] Graphiti Home, http://www.eclipse.org/graphiti/

[26] K. Czarnecki and S. Helsen. 2006. Feature-based survey of model transformation approaches. IBM Syst. J. 45, 3 (July 2006), 621-645. DOI=10.1147/sj.453.0621 http://dx.doi.org/10.1147/sj.453.0621

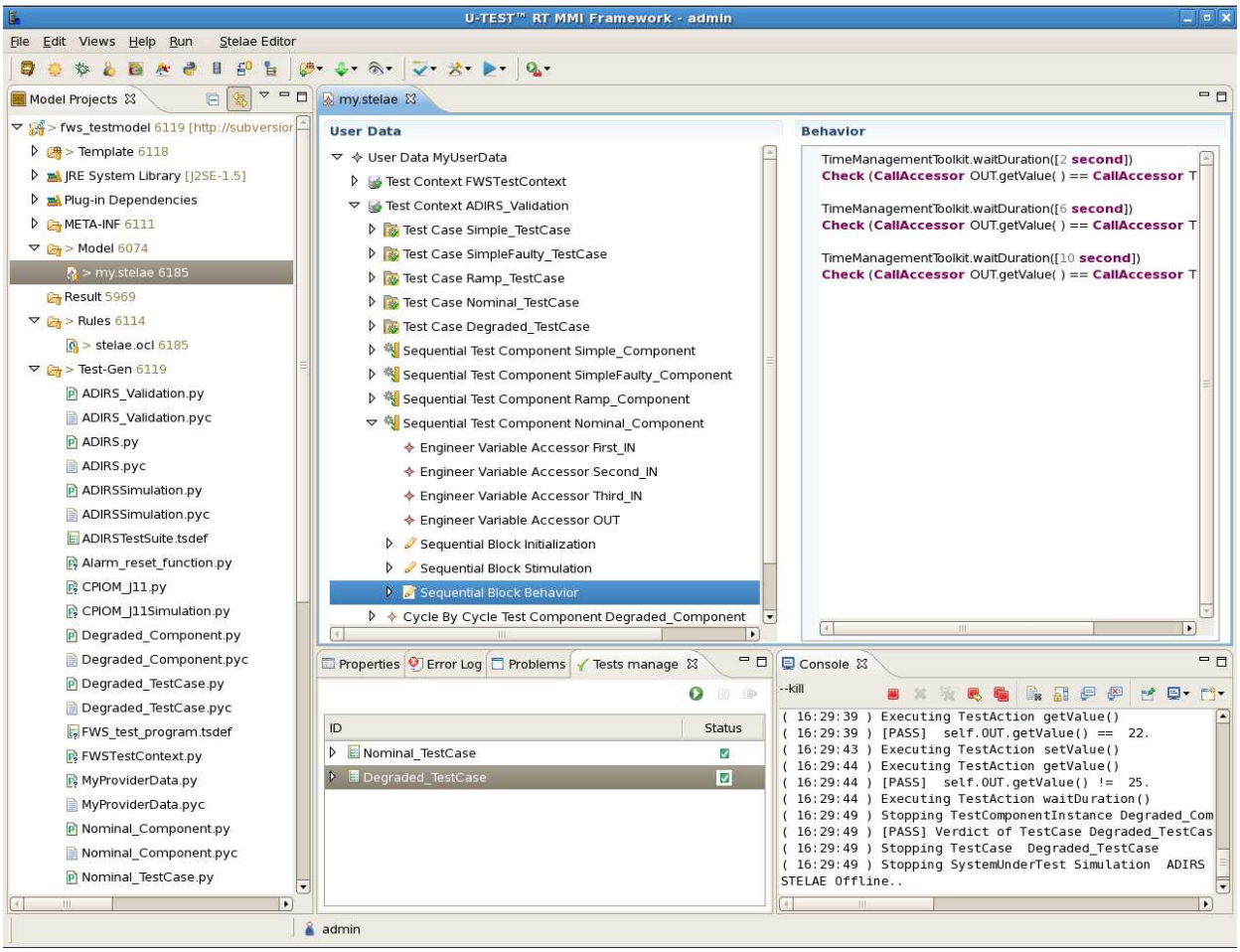

Figure 8.a. "STELAE" Perspective

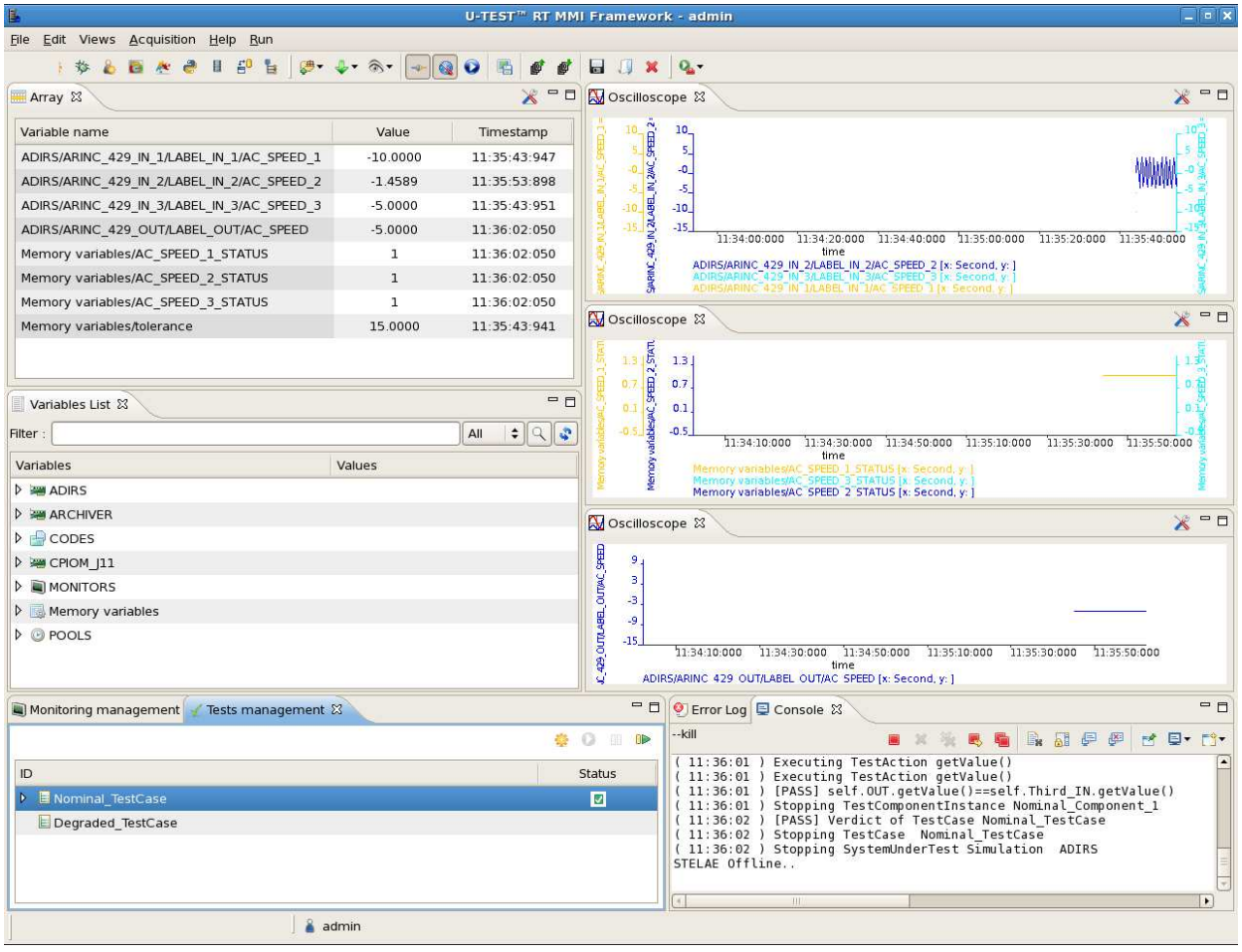

Figure 8.b. "Runtime" Perspective