Softw Syst Model (2019) 18:1543–1570
https://doi.org/10.1007/s10270-017-0620-y

REGULAR PAPER

From use case maps to executable test procedures: a scenario-based approach

Nader Kesserwan[1] · **Rachida Dssouli**[1] · **Jamal Bentahar**[1] · **Bernard Stepien**[2] · **Pierre Labrèche**[3]

Received: 23 June 2016 / Revised: 29 July 2017 / Accepted: 16 August 2017 / Published online: 31 August 2017

Abstract Testing embedded systems software has become a costly activity as these systems become more complex to fulfill rising needs. Testing processes should be both effective and affordable. An ideal testing process should begin with validated requirements and begin as early as possible so that requirements defects can be fixed before they propagate and become more difficult to address. Furthermore, the testing process should facilitate test procedures creation and automate their execution. We propose a novel methodology for testing functional requirements. The methodology activities include standard notations, such as UCM for modeling scenarios derived from requirements, TDL for describing test cases and TTCN-3 for executing test procedures; other test scripting languages can also be used with our methodology. Furthermore, the automation of the methodology generates test artifacts through model transformation. The main goals of this test methodology are to leverage requirements represented as scenarios, to replace the natural language test case descriptions with test scenarios in TDL, and to generate executable test procedures. Demonstration of the feasibility of the proposed approach is based on a public case study. An empirical evaluation of our approach is given using a case study from the avionics domain.

Keywords Model-driven testing · Testing methodology · Embedded systems · Test generation · TTCN-3 · TDL · UCM

Communicated by Professor Jean-Marc Jezequel.

✉ Nader Kesserwan
n_kesse@encs.concordia.ca

Rachida Dssouli
rachida.dssouli@concordia.ca

Jamal Bentahar
bentahar@ciise.concordia.ca

Bernard Stepien
bernard@eecs.uottawa.ca

Pierre Labrèche
Pierre.Labreche@cmcElectronics.ca

[1] Concordia Institute for Information Systems Engineering (CIISE), Concordia University, Montreal, Canada

[2] University of Ottawa, Ottawa, Canada

[3] Esterline CMC Electronics, Montreal, Canada

1 Introduction

As software systems become increasingly complex, the demand on software verification grows. Testing software systems in regulated industries is very expensive, as the testing process must be based on a rigorous testing methodology to comply with stringent objectives; testing often requires the integration of various components: the System Under Test (SUT), simulators of end-systems, measurement systems, and a test management system. To address a growing demand, several new technologies have emerged to help with the development and verification of high-quality systems.

For a safety-critical system, adopting a suitable test methodology is imperative to ensure the effectiveness of verification for certification. For instance, the aeronautics industry has witnessed an exponential growth of software in embedded systems, military and civil. For example, in a 40-year period, the portion of functionality provided to military aircraft rose from 8% in 1960 (F-4 Phantom) to 80% in 2000 (F-22 Raptor). The F-22A is reported to have 2.5 million lines of code [10]. The growing complexity of flight software presents many challenges to the current test

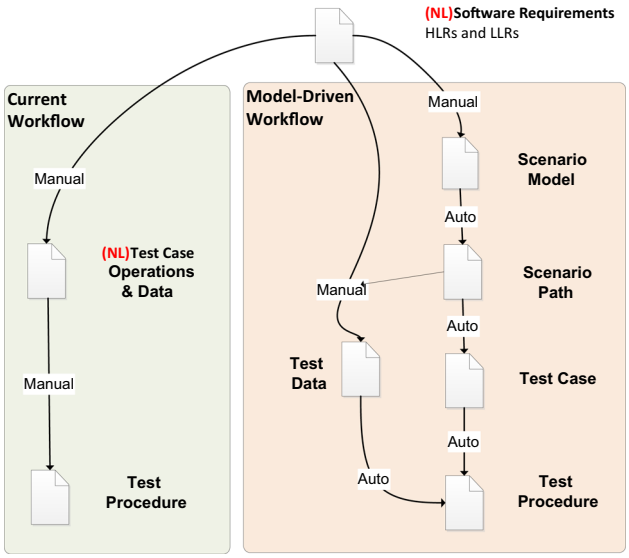


Fig. 1 Current versus model-driven testing workflow

technology [22]. While business objectives are focused on time and costs, process requirements are set very high by the certification authorities. To address concerns with evolving software technologies, the Radio Technical Commission for Aeronautics (RTCA) [18] published an updated guidance document, DO-178C "Software Considerations in Airborne Systems and Equipment Certification" [9]. The document defines process objectives for five design assurance levels (A to E) to ensure an adequate quality of software for airworthiness. The software verification process objectives ensure a high degree of confidence that software executes as intended on the target platform.

The software requirements in the current practice (left side of Fig. 1) are expressed in natural language (NL) and are layered as high-level requirements (HLR) and low-level requirements (LLR) in separate artifacts. Requirements are subsequently used as the basis, along with test engineer knowledge (implicit), for writing test cases (TCs) in NL, and then manually developing executable test procedures (TPs).

Our goal is to define a model-driven testing methodology (right side of Fig. 1) that supports test automation based on requirements of the SUT, formalizing each TC in a TP (also called test description) independently of its implementation, and finally translating each TC to a TP in a test scripting language.

The basic motivation for describing requirements into scenario models is to support the auto-generation of less-technical TCs and then auto-generation of executable TPs. A validation of the methodology is conducted by studying the effectiveness of generated test cases on industrial case study, in terms of path coverage.

In modern industrial software development, scenarios are widely used both for early requirements elicitation and for requirements specification [3]. A scenario defines a sequence of events comprising stimulus, actions and expected responses related to the behavior of the component or SUT. Collections of scenario models must address both *normal range* and *robustness* requirements. The system and software requirements, in particular functional and operational requirements, are analyzed and described into scenario models; from these scenarios will eventually derive scenario paths, test descriptions and TPs. When software requirements are modeled with use cases, a scenario can take a specific path through the model to instantiate a use case. Given a coverage criteria of the test model, such as path coverage, the behavioral test model hereby enables structured derivation of all possible test scenarios for the selected use case. The execution of these scenario paths on the SUT, when refined with sufficient details, can produce measurements of the extent to which a criterion is satisfied in terms of coverage. Consequently, it helps spotting gaps in the use case that may reveal the need for additional test scenarios to satisfy the criterion.

This paper is organized as follows: Sect. 2 provides background information. Section 3 presents the approach, which is followed by a feasibility of this approach in Sect. 4. An evaluation of the approach is presented in Sect. 5. Section 6 surveys related work and Sect. 7 concludes and draws future work directions.

2 Background

Testing is a major cost factor during software development, sometimes consuming more than 50% of the overall development effort [36] and [11]. Several methods and techniques have been developed to streamline the testing process. One of those techniques is model-driven testing [2] (MDT), which is an automation of model-based testing (MBT) that uses model transformation technology on formal models, their metamodels, and transformation rules defined in terms of mappings between the elements of metamodels. Model-driven architecture is a development paradigm that has been proposed 15 years ago as "an approach to IT system specification that separates the specification of system functionality from the specification of the implementation of that functionality on a specific technology platform" [32]. In this paradigm, models at various levels of abstraction are the central software design artifact. They are used to facilitate both abstraction and automated development. According to a 2011 survey in the car industry [35], "Model-based testing (i.e. the generation of test cases out of a test model) is currently not used intensively. Only 35% of the participants use it right now, but almost 50% plan to use it in the near future." Requirements described in models enable the automatic generation of tests. Requirements-based testing (RBT) has been applied in the testing of complex software systems. As early as 1985, RTCA

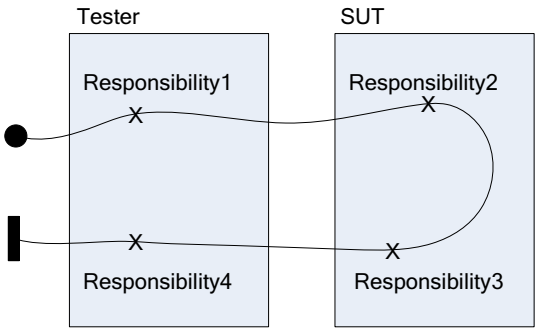


Fig. 2 UCM core notation

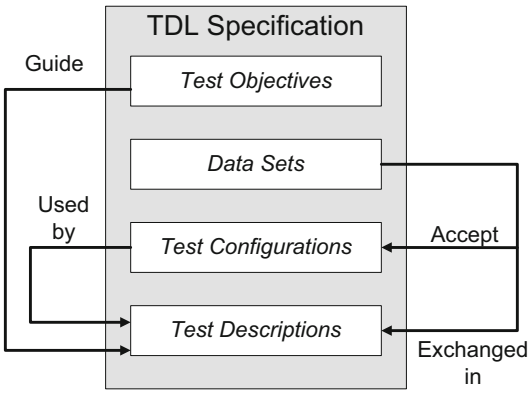


Fig. 3 Major parts of a TDL Specification

DO-178A [8] introduced RBT, traceability from requirements to tests, and test coverage. RBT demonstrates that the software or the system meets the requirements. The RBT approach offers several advantages to test designers such as a precise and detailed description of the system's functionality and constraints, and the possibility to identify tests from the requirements context and to design them as black-box tests.

There are several approaches to organize and specify requirements such as IEEE Standard 830-1998 SRS templates [21]. A TC is a high-level specification for the TP. Numerous languages can be used to implement TPs. We introduce the following notations to capture functional requirements in terms of causal scenarios, describe the software TCs as scenarios and implement TPs and execute them against SUT:

i. **Use Case Maps (UCM):** a visual notation for describing, in a high-level way, how the organizational structure of a complex system and the emergent behavior of that system are intertwined. UCM [7] as part of the User Requirements Notation standard [23] was suggested to represent the behavior of a system as a visual use case, i.e., a scenario model. UCM is a scenario-based notation enabling the description and analysis of use cases and scenarios. It has been used to capture functional requirements in terms of causal scenarios composed of responsibilities that can be attached to underlying abstract components. Engineers can use tools such as jUCMNav [14] to create, maintain, and transform UCM models. The core notation of UCM has the following fundamental elements, represented in Fig. 2: (1) a *causal path* represented by a wiggly line; (2) *rectangular boxes* that represent components (Tester and SUT); and (3) *responsibility points*? bound to components along the path? that represent actions or functions to be performed.

The *responsibilities* elements in UCM are abstract and can represent actions or tasks to be performed by the components. The components themselves are also abstract and can represent software entities (objects, processes, network entities, etc.) as well as non-software entities (e.g., users, actors, processors). The interested reader can consult the UCM notation [7] and [23] or the UCM Quick Reference Guide [19] for more information.

ii. **Test Description Language (TDL):** TDL [15] is a standardized scenario-based approach proposed by the European Telecommunications Standards Institute (ETSI) to describe software test cases as scenarios. TDL is a new language created for specifying "formally defined *Test Descriptions* used as the starting point for test automation. It allows describing scenarios on a higher abstraction level than programming or scripting languages. Furthermore, TDL can be used as an intermediate representation of tests generated from other sources, e.g., simulators, test case generators, or logs from previous test runs." [34]. TDL is a general formal language for representing *Test Descriptions* which are used mainly for communication between stakeholders as the basis for implementing concrete tests. The TDL design is centered on three separate concepts: (1) the metamodel principle that expresses its abstract syntax; (2) concrete syntax, which is user-defined for different application domains; and (3) the TDL semantics that can be found in metamodel elements. The main TDL structure elements expressed in *italic* throughout the paper are shown in Fig. 3.

(a) A *Test Objective* that states the reason for designing either a *Test Description* or a particular behavior of a *Test Description*. It can be written as a simple text in natural language.
(b) A set of typed *Data Sets* used in the interactions between components in a *Test Description*;
(c) A *Test Configuration,* which is a set of interacting components (Tester and SUTs).
(d) A set of *Test Descriptions* to describe one or more test scenarios based on the interactions of data exchanged between the Tester and the SUTs. The control flow of

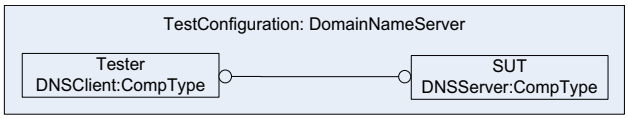


Fig. 4 TDL *Test Configuration* element

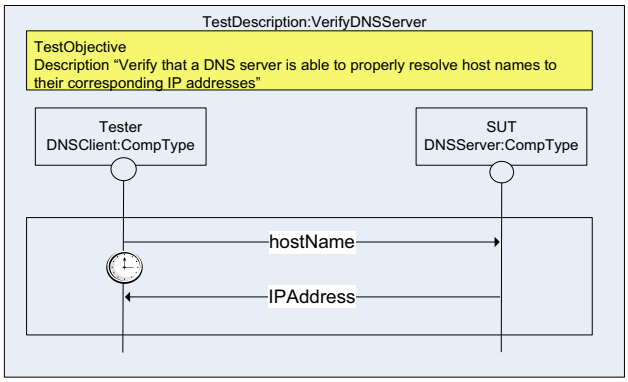


Fig. 5 TDL *Test Description* element

a *Test Description* is expressed in terms of the composition of operations such as sequential, parallel, alternative, iterative.

The TDL elements are explained with an example based on the Internet's Domain Name System (DNS) that aims at verifying that a DNS server is able to properly resolve host names to their corresponding IP addresses. The *Test Configuration* element that is composed of a set of two interacting components is shown in Fig. 4.

The *Test Description* depicted in Fig. 5 represents the expected behavior based on the *Test Objective* and expresses the test in terms of exchanged *Data Set* instances (*hostName* and *IPAddress*) between the two components.

Interested readers can refer to [34] and [26] that discuss the application of TDL to a number of common application scenarios.

iii. **Testing and Test Control Notation (TTCN-3):** a standard language for test specification that is widespread and well established. TTCN-3 [16] was selected for its industrial strength to implement and execute TPs against SUT. TTCN-3 is a test specification language developed by ETSI that is applicable to a variety of application domains and levels of testing. It is designed for specifying collections of test cases in Abstract Test Suites that are then used to test the SUT. The top-level unit of TTCN-3 is a module that corresponds to a compilation unit in traditional programming languages. The module may contain a single case or several test cases that can be compiled or interpreted. A test case can be seen as the main function of a single case; it is always executed within an entity called a component in order to express its behavior. The result of executing a test case is a verdict that determines if the SUT has passed the test. Listing 1 shows a test case that implements the DNS example.

```
1.  testcase VerifyDNSServer() runs on DNSClient {
2.  template String hostName := "MyHostName";
3.  template String IPAddress:= "192.124.135.56";
4.   clientPort.send(hostName);
5.   DNSTimer.start(10.0);
6.   alt {
7.      [] clientPort.receive(IPAddress) {
8.         setverdict (pass); }
9.      [] clientPort.receive {
10.        setverdict (fail); }
11.     [] DNSTimer.timeout {
12.        setverdict(fail) } }
13. }
```

A component should be defined (DNSClient) with a single port (clientPort) to communicate with the DNS server (SUT). The clientPort sends a data instance (*hostName*) to the SUT (line 4). Directly after, a timer is started (line 5) and set to run for 10 s. If the clientPort receives (line 7) the expected data instance (*IPAddress*) within 10 s, the test case passes. If the clientPort receives anything other than *IPAddress* (line 9) or the *DNSTimer* times out (line 11) the test case fails.

3 The scenario-based approach

The forward methodology shown in Fig. 6 starts when the test designer wants to describe the NL requirements into UCM scenario models.

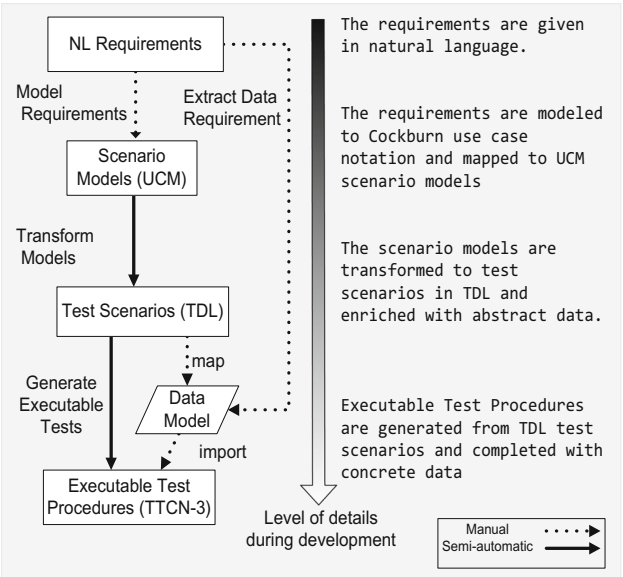


Fig. 6 MDT methodology for scenario-based testing

The key points of the testing methodology are: (1) NL requirements are described in UCM scenario models; (2) these models are transformed semiautomatically to test scenario descriptions in TDL that are completed manually with test objectives and data instances; and (3) the obtained test scenarios are used along with concrete test data as the basis to generate executable TPs in TTCN-3.

The approach can be seen as a process of successive refinements of specifications that involves model transformation and the insertion of additional information. The testing methodology must ensure test effectiveness—all requirements are covered—while also aiming for test efficiency—the testing effort is reduced by decreasing the manual development while ensuring the discovery of implementation errors in the SUT. Compared to the baseline process, it offers features that should be attractive to test designers, such as scenario coverage and a simple structure, where ease of use and understandability are key. Furthermore, it supports test automation by generating executable TPs from test scenarios. The integration of UCM scenarios into the existing development process, such as the requirements' process, should only require minimal effort and adjustments.

In this paper, a use-oriented technique is used for software requirement analysis. Use cases and usage scenarios increase system understandability for developers and improve communication between system stakeholders.

In the following subsections, we explain how model transformation and insertion of additional information are performed at each step in the process. A case study is conducted in Sect. 4 to demonstrate the feasibility of the approach.

3.1 Describing NL requirements into UCM models

In order to facilitate the modeling of the NL requirements into UCM elements, the requirements are written in Cockburn use case notation [1]. With some basic knowledge of the jUCMNav tool, the modeled use case is mapped manually to UCM scenarios models. The concrete metamodel of the UCM notation is shown in "Appendix 5."

The scenario models represent the system from a functional execution sequence perspective, which is another form in which to represent the system and software requirements. Scenarios provide benefits for system comprehension, design, testing and maintenance. Scenarios can be grouped, related and decomposed for better management, reusability and analysis. Furthermore, scenarios can be used later in the verification process to drive the test specification and to direct the development of TPs.

In DO-178C, the software verification process defines activities for determining that the software aspects of airborne systems comply with airworthiness requirements. One of the activities defined in the process is to verify that the system requirements allocated to software have been developed into HLR that satisfy those system requirements. Trace data are generated to support this verification.

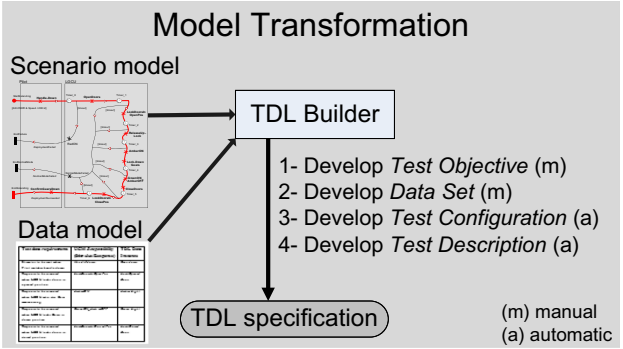


Fig. 7 TDL builder process

A relationship between each unique system-level requirement and its embodiment in the software requirement is created, allowing traceability between software requirement and HLR. This relationship should allow for bidirectional traceability, meaning that the traceability chains can be traced in both the forwards and backwards directions.

In our approach, UCMs are an intermediate step toward writing *Test Descriptions*.

3.2 Transform UCM scenario models to Test Description in TDL

A UCM scenario model conveys information to help to develop some of the TDL specification elements, in particular, *Test Objective*, *Test Configuration* and *Test Description*.

Since UCM scenarios deal only with behavior, the concept of data is yet to be supported. Therefore, a data metamodel is developed (shown in "Appendix 6") that is based on test data requirements to help identify UCM *responsibilities* that exchange messages, develop the TDL *Data Sets* and detail the TTCN-3 data with concrete values.

Next, a process is developed called *TDL Builder* as shown in Fig. 7, to transform the UCM scenario model and data model (additional information) into a complete and valid TDL specification.

The *TDL Builder* process transforms the UCM scenario to four TDL elements. The development of each element is shown in the following:

A. Test objective

Several *Test Objectives* can be developed by analyzing the scenario model. *Test Objectives* set guidelines to design the *Test Description* or to design a particular behavior. Typical UCM objects include *component, responsibility, comment, timer,* and*condition*. The *Test Objective* can be enriched by adding additional information from the system requirements.

B. Data set

Table 1 Test data for UCM scenario

Test data requirements	UCM responsibility (Stimulus/response)	TDL data instance
Stimulus/response to be exchanged	Interaction	Data instance

In general, the Test Inputs for the TPs are produced in the test analysis and design process (not shown here). We assume that it is possible to select enough data from the analysis process to enable the development of Test Input for use in the TPs.

A *responsibility* definition in UCM represents an action or the steps to perform, either informally through its name or more formally with the help of its expression. Using this information, the *responsibilities* involved in a stimulus/response action can be flagged as interaction messages and mapped into *Data Instances* in TDL. A data model based on data requirements composed of three levels of test data abstraction is developed:

a) **Stimulus/response:** a subset of test data requirements can be represented abstractly as I/O message in UCM *responsibility* objects;
b) **Test data scenario:** the I/O messages in the stimulus/response level are developed into a TDL *Data Sets*.
c) **Test data procedure:** The *Data Sets* are developed using templates.

Table 1 shows three columns of test data: the test data requirements, the complete set of UCM *responsibilities* and its corresponding TDL *Data Instances*.

Each UCM responsibility in the second column (interaction) is either a stimulus to send or a response to receive. This interaction is represented as a TDL *Data Instance* in the third column.

The *Data Instances* to be used in the *Test Description* are developed manually and grouped in *Data Sets.* They are an abstract representation of the corresponding data-related concepts in a concrete-type system.

C. Test Configuration and Test Description

A *Test Configuration* in TDL specifies the communication infrastructure necessary to build upon the*Test Description*. As such, it contains all the elements required for the exchange of information, such as *Component Instances* and *Connections*. Each *Component Instance* specifies a functional entity of the test system. A *Component Instance* may either be a part of a Tester, or a part of an SUT. The *Test Configuration* element consists of:

– A Tester;
– SUT components;
– A gate;[1] and
– Interconnections between Tester and SUT components via gate instance.

The metamodel of *Test Configuration* and *Test Description* [15] is shown in "Appendix 4."

The *Test Description* in TDL defines the expected behavior, the actions and the interactions between system components. The *Test Description* is associated with exactly one *Test Configuration*, and may be associated with any number of data elements that represent the formal parameters. Any number of *Test Objectives* can be attached manually to the *Test Description* to help to specify its design.

The *Test Description* in TDL defines the test behavior based on ordered atomic or compound behavior elements. A *responsibility* object in a UCM scenario model represents an action to be performed by its enclosing component. Its equivalence in TDL is found in the *Action Reference* element, which is an atomic behavior used to refer to an *Action* element to be executed. The dynamic and static *stub* objects that contain sub-maps are not mapped to any TDL element. However, the *responsibilities* elements, enclosed in these objects, can be mapped to corresponding ones in TDL. An *Action Reference* may have a *Component Instance* attribute identifying the component instance on which the action is to be performed. Any information exchanged via the gates is represented abstractly and can be referenced by TDL *Interaction* elements. An interaction can represent a message sent from a source and received by a target.

In our approach, we used the feature path traversal algorithm in the jUCMNav tool to export UCM scenario models in XMI format [5]. We developed a java tool to parse the exported scenario and transform it automatically to TDL *Test Configuration and Test Description* elements. The exported scenarios are structured by a metamodel, see "Appendix 6," and as such can be handled by model transformation. The exported scenarios have exhaustive coverage of the UCM model. The algorithm uses a depth-first traversal [24] of the scenario that captures the UCMs' structure. The algorithm traverses the path elements beginning at a start point until a stop point (AND-join, waiting place, or timer) is reached. Then, the algorithm backtracks to get the next available branch of an AND-fork (unvisited branches) or the next start points if any. The traversal is successful if all elements along the path are marked as visited. The algorithm can prevent infinite loops through a maximum number of visits. The exported scenario contains traversed UCM elements such as *Component Instance, Gate Instance, Action Reference, Interaction* that we use to develop the TDL specification that can be compiled in the proposed TDL concrete syntax.

[1] A gate is a point of communication for exchanging information between components, and it specifies also the data that can be exchanged.

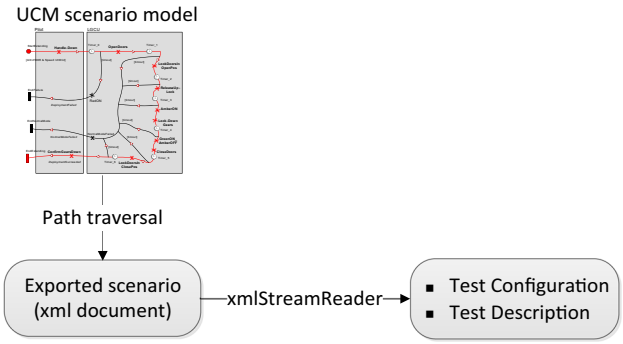


Fig. 8 The development of TDL *Test Configuration* and*Test Description*

The java tool parses the exported scenario using XMLStreamReader interface and automatically generates the two TDL elements *Test Description* and *Test Configuration.*

The interface XMLStreamReader is used to iterate over the various events in the exported scenario to extract the information and convert it to TDL syntax. Once we are done with the current event, we move to the next one and continue till end of the scenario. The events can be for example the start of element, the end of element or attribute.

Figure 8 illustrates the development of the *Test Configuration* and *Test Description* from UCM scenario.

The transformation algorithm from UCM to TDL generates only linear scenarios or one alternative per scenario, while a typical TP has alternative responses. Therefore, the exported scenarios can be manually merged by using common behavior until a fork is encountered that produces different responses to the tester. The post-processing of linear scenarios was recommended in [5].

Finally, the resulting elements can be combined along with *Test Objectives* and *Data Sets* in one TDL specification and used a TDL Editor[2] to edit and validate the specification. The Editor defines the specific domain of the TDL language and is based on its metamodel. The Domain Specific Language (DSL) of the TDL is written in the *Xtext* language development framework [17].

We made the java tool and TDL Editor available online;[3] interested readers can download the eclipse project to generate TDL *Test Configuration* and*Test Description* elements from UCM scenarios. The TDL specification is based on the TDL metamodel [15] and expressed in concrete syntax. It clearly separates the abstract test scenario from its associated executable TP by providing an abstraction level. As a result, the test designer can focus on describing a test scenario that covers the given *Test Objectives* rather than fully implementing the script. It is the final implementation as an executable TP that will ensure the discovery of implementation errors in the SUT.

[2] Obtained from Philip Makedonski, University of Göttingen.

[3] https://users.encs.concordia.ca/~bentahar/Model_Transformation/.

The *Annotation* element in TDL is a means to attach user-defined semantics to any element of a TDL model, and it provides a pair of the key/value attributes. In our approach, we have trace data that shows the HLR are traceable to software requirements and that the LLR are traceable to HLR. The test scenarios are traced indirectly (via UCMs) to the HLRs and LLRs. The executable TPs are traced to the abstract test scenarios in TDL. Therefore, compliance with DO-178C standard is achieved for the traceability objective. LLRs are developed from HLRs, and as defined by DO-178C, an association between a requirement and its related items is necessary. The TDL can be produced from UCMs developed from HLRs or from LLRs: the methodology is applicable to HLR- or LLR-based testing.

3.3 The transformation of TDL specification to TTCN-3 modules

The derivation of TDL specifications from a UCM scenario model is an abstraction of the expected behavior between components and cannot be used directly on the actual SUT. The abstract test scenarios thus described lack concrete details about the SUT and its environment. Therefore, executable TPs should be derived and sufficiently detailed with test data and interface requirements (additional information) to correctly communicate with the SUT. We propose to use TTCN-3 language to implement the abstract test scenarios defined by the TDL specification package. The document TTCN-3 Core Language [20] defines the syntax of TTCN-3 using extended BNF.

One of the design objectives of TDL is to be less technical and thus user friendly for non-technical users and that it can serve as the basis for the implementation of executable tests that are by definition highly technical.

The TDL test specification is transformed into executable TPs by applying structural transformation, e.g., a TDL element, shown in *italic* across the paper, is transformed into a TTCN-3 module. Therefore, we consider that an executable test suite in TTCN-3 is broken down into four types of modules: (1) a Test Configuration module that consists of a set of inter-connected test components with well-defined communication ports, (2) a Test Description module which usually contains behavioral program statements that specify the dynamic behavior of the test components over the communication ports, (3) a Test Oracle module that contains templates (expected result or responses) used to test whether a set of received values matches the template specifications, and (4) Test Input module that contains input data (stimulus) to be transmitted to the SUT. This modular approach of deriving the executable TPs facilitates the model transformation between TDL and TTCN-3 and promotes the reusability of the generated modules.

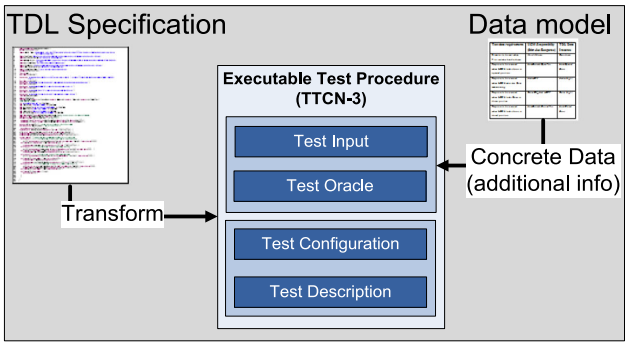


Fig. 9 Derivation of executable TP in TTCN-3

Deriving the TTCN-3 modules that compose the executable test suite is done semiautomatically by applying transformation rules defined between TDL and TTCN-3 (see "Appendix 3"). These rules are programmed and implemented in a tool based on model-to-text technology called *Xtend.* We made the tool available online.[4] Fig. 9 shows the derivation of the executable test suite in TTCN-3.

In our approach, the TDL elements developed previously were used, based on transformation rules, to derive the corresponding modules in TTCN-3. Next, the derived Test Input and Test Oracle modules were enriched manually with concrete data from the data model to enable the execution of the TPs. The development of the TTCN-3 modules is discussed in the following:

A. The development of the Test Input and Test Oracle

As mentioned earlier, TDL does not offer a complete data type system. Instead, it depends on *Data Set* elements—whose *Data Instances* are an abstract representation of the corresponding data-related concepts in a concrete-type system. Therefore, the *Data Instances* developed in the previous section can be used along with data requirement analysis to develop concrete data definition. In our approach, a TTCN-3 data module that contains Test Input and Test Oracle definitions based on *Data Sets* is developed. The language *Xtend*, part of the Eclipse *Xtext* project, is used to generate partial TTCN-3 code from TDL *Data Set* syntax. All *Data Set* instances can be identified from the parsed TDL model and generate a record type for each in TTCN-3 syntax, see the description of ***Rule# 22*** and ***Rule# 23*** in "Appendix 3" for more details. After the TTCN-3 data module is partially generated and test data becomes available, the module is completed with Test Oracle information and typed with concrete TTCN-3 types.

B. The development of the Test Configuration

When describing a *Test Configuration* in TDL, the main focus is usually on the test components and their communication, whereas an executable test requires a more detailed configuration. A *Test Configuration* in TDL consists of Tester and SUT components, gates, and their interconnections represented as the *Connection*. A TTCN-3 configuration should consist of a set of inter-connected test components with well-defined communication ports and an explicit test system interface which defines the boundary of the test system. Furthermore, the communication between components is achieved via well-defined port types such as message-based and procedure-based ports. To enable the transformation of an abstract test scenario to a concrete and executable TP, we developed transformation rules; see "Appendix 3," based on both language elements. The TDL *Test Configuration* contains the necessary objects, test components and communication channels to build the TTCN-3 configuration module. The concrete details needed to communicate correctly with the SUT, such as the message type to be sent or received, are imported from the TTCN-3 data module where the Test Inputs and Test Oracle are defined. The TDL *Test Configuration* components such as gate, Tester, and SUT are transformed to equivalent objects in TTCN-3 by applying ***Rule# 2***, ***Rule# 3***, and ***Rule# 4***, respectively. More specifically, these rules are implemented in our tool that iterates over the TDL model to collect all *Gate Type* elements and generates for each a message-based port statement in TTCN-3 syntax. The *instantiate* elements are parsed to generate a component-type statement with an associated port.

C. The development of the Test Description

The TDL *Test Description* defines the test scenario behavior, mainly composed of the actions and interactions exchanged between components over the communication gates. An action is used to specify a procedure (e.g., local computation, function call, physical setup) in an informal way, whereas interactions refer to the data being exchanged between the components. In TTCN-3 realization, our tool iterates over the TDL model elements to parse the behavior elements and generate equivalent statements for each in TTCN-3.

Our tool parses the *sends instance* statements (interaction) and generates a TTCN-3 message statement (***Rule# 8***). The *action* statement is parsed to generate a function signature and a function call (***Rule# 12***). The obtained function is refined at the TTCN-3 level when applicable.

Other TDL behavioral statements are mapped to TTCN-3 constructs to be used in TPs or in functions by applying the corresponding rule.

3.4 The completeness and soundness of the model transformations

In general, model transformations are used between different domains for model evolution, code generation and analysis.

[4] https://users.encs.concordia.ca/~bentahar/Model_Transformation/.

The UCM models are adequate for describing the functional requirements of a system. Their automated transformation to TDL models bridged the gap with the executable TPs in TTCN-3. However, the metamodel of the exported scenario generated from UCM scenario, early in the process, doesn't have alternative element that normally a TP has to handle alternate test behavior. The absence of alternative element in the scenario metamodel required manual adjustment of the generated TDL models to merge those that constitute alternate test behaviors. The transformation of TDL models allowed refining and generating executable TPs that can be performed on the SUT. The model transformations here link the various test artifacts and promise to reduce the required amount of manual work for test development.

4 Approach feasibility

The feasibility of the approach is demonstrated via a case study from the avionics public domain called landing gear system (LGS) [4].

The LGS specifications are categorized into functional, safety and timing requirements. In the next sections, the behavior of the LGS is described from a Pilot's perspective, formalized into a given use case notation and then mapped to UCM scenario models. The LGS supports an aircraft when it is on the ground, allowing it to take off, land and taxi. Most modern aircraft have a retractable undercarriage, which folds away during flight to reduce air resistance or drag. A conventional hydraulic LGS has a tricycle configuration consisting of the nose and the main (left and right) landing gears. Each landing configuration contains a door, the landing gear and the associated hydraulic cylinders. The LGS is representative of critical embedded systems. Failure to deploy it puts the life of passengers in danger and causes massive airframe damage upon landing. Prior to landing, the landing sequence of an aircraft is: open the landing gear box doors, extend the landing gear and close the doors. After taking off, the Retraction Sequence is: open the landing gear box doors, retract the landing gear and close the doors. The LGS is composed of: (a) mechanical part; (b) digital part; and (c) a Pilot interface part which is further detailed in the next paragraph in order to identify the requirements. For more information about parts (a) and (b), please refer to [4].

The Pilot commands the retraction and extension of the gears by switching a handle up or down. When the handle is switched to "Up" the retracting landing gear sequence is executed, and when the handle is switched to "Down," the landing gear extension sequence is executed. Additionally, the Pilot's control panel has a set of lights indicating the current positions of the gears and doors, as well as the current health state of the system and its equipment. These lights and their indications are:

- One green light: "gears are locked down".
- One amber light: "gears are in transition".
- One red light: "landing gear system failure".
- No light is ON: "gears are locked up".
- Doors locked opened sign is ON: "all doors of the landing gear boxes are locked in opened position".
- Doors locked opened sign is OFF: "all doors are unlocked".
- Doors locked closed sign is ON: "all doors of the landing gear boxes are locked in closed position".
- Doors locked closed sign is OFF: "all doors are unlocked".
- Normal Mode Fail sign is ON: "Normal Mode Fail".
- Normal Mode Fail sign is OFF "Normal Mode Pass".

The expected behavior of the LGS is implemented by the control software whose aim is twofold: (1) control the hydraulic devices according to the Pilot's orders and to the mechanical devices' positions and (2) monitor the system and inform the Pilot in case of any malfunction.

Before showing how the functional and timing requirements of the LGS can be captured by UCM scenario models, the LGS requirements are formalized as described next.

4.1 Modeling LGS requirements into Cockburn use case notation

The LGS requirements fall into two basic scenarios: the Extending Sequence and the Retraction Sequence. For the purpose of clarification, the Extending Sequence scenario, as defined in the case study, is used as running example. Next, consider that the Pilot wants to land the airplane and so switches the handle down when the aircraft has an indicated airspeed of less than 200 *knots* and an altitude less than 2500 *feet.* The Extending Sequence scenario written as a use case follows:

USE CASE: Extending Sequence.
***Primary Actor*:** Pilot
Secondary Actor: Landing Gear Control Unit (LGCU)
Scope: LGS.
Precondition: Airspeed is less than 200 *knots* and altitude is less than 2500 *feet.*
Minimal guarantee: Landing gears are extended in emergency mode.
Success guarantee: Landing gears are extended in normal mode.
Trigger: Pilot switches handle down.

Main success scenario:

1. Pilot switches handle down and it stays down.
2. LGCU activates doors opening.
3. LGCU locks door in opened position.
4. LGCU switches doors locked open sign to ON

5. LGCU releases up-lock gears.
6. LGCU switches amber light to ON.
7. LGCU locks down gears when they reach the full-down position.
8. LGCU switches green light to ON and amber light to OFF.
9. LGCU activates doors closing.
10. LGCU locks door in closed position.
11. LGCU switches doors locked closed sign to ON
12. Pilot confirms successful deployment of the landing gears.

Extensions: (Failure mode)

1.a If the landing gear command handle has been DOWN for 15 s and the gears are not locked down after 15 s, then the LGCU switches red light to ON (failure in deployment).
2.a If one of the three doors is still seen locked in the closed position more than 7 s after activating doors opening, then the LGCU fails Normal Mode.
3.a If one of the three doors is not seen locked in the opened position more than 7 s after activating doors locking in opened position, then the LGCU fails Normal Mode.
5.a If one of the three gears is still seen locked in the up position more than 7 s after releasing the up-lock, then the LGCU fails Normal Mode.
9.a If one of the three gears is not seen locked in the down position more than 10 s after releasing the up-lock, then LGCU fails Normal Mode. If one of the three doors is still seen locked in the opened position more than 7 s after activating doors closing, then the LGCU fails Normal Mode.
10.a If one of the three doors is not seen locked in the closed position more than 7 s after activating doors locking, then LGCU fails Normal Mode.

Next, we proceed with the mapping of the Extending Sequence use case to UCM scenario models.

4.2 Mapping LGS use case to UCM scenario models

UCM scenario models can be built by mapping the actors and the actions elements defined in the Extending Sequence use case. The mapping is straightforward, for example, the Primary Actor (Pilot) and the Secondary Actor (LGCU) are mapped manually to two UCM components: *Pilot* and *LGCU*. The actions to be performed by each component, such as *Handle_Down* and *ReleaseUp_Lock* are allocated to UCM *responsibility* elements. As a rule, each action in the use case is mapped to one *responsibility* element in UCM. As a result, two lists of *responsibility* are extracted from the use case and bound to their corresponding components:

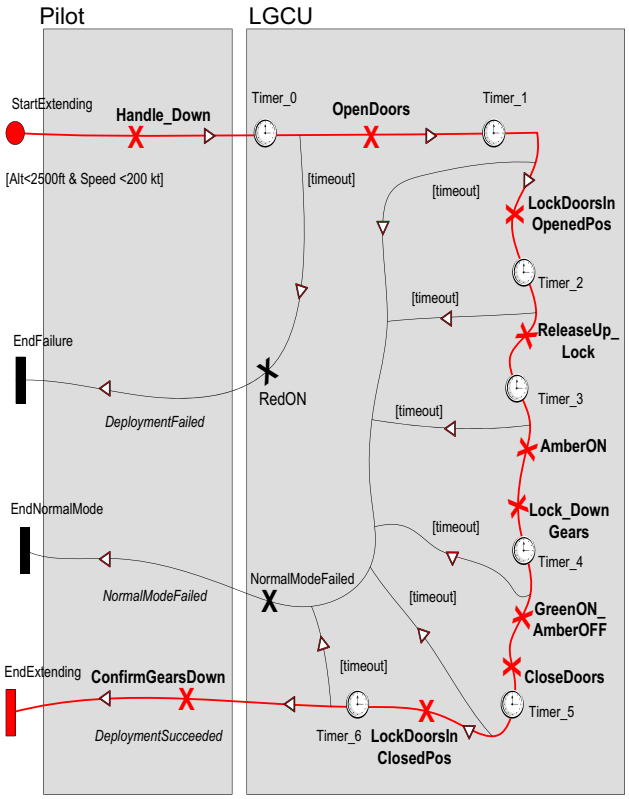


Fig. 10 UCM scenario models built from an Extending Sequence use case

- *Pilot*: { *Handle_Down* and *ConfirmGearsDown*}
- *LGCU*:{*OpenDoors, LockDoorsInOpenedPos, Release Up_Lock, AmberON, Lock_DownGears, GreenON_Amb erOFF, CloseDoors, LockDoorsInClosedPos, RedON,* and*NormalModeFailed*}.

With some basic knowledge of the jUCMNav tool, the two lists of *responsibility*; *Pilot* and *LGCU,* along with timed requirements in the use case can be modeled into UCM scenarios. Figure 10 shows a UCM map that is composed of two components with their bounded *responsibilities*. The time constraints and functional requirements are modeled as indicated by the Extending Sequence use case. The map in Fig. 10 encloses eight possible scenario models representing the Extending Sequence requirements of the LGS.

These scenario models fall into three major groups:

a) Successful Deployment Group: contains one scenario model, labeled [*DeploymentSucceeded*].
b) Gears Deployment Failed Group: contains one scenario model, labeled [*DeploymentFailed*].
c) Normal Mode Failed Group: contains six scenario models, all of them end in the path labeled [*NormalMode-Failed*].

The execution of any of the scenario models begins at the *StartExtending* point (filled circle) and terminates in one of the three End points (bars); *EndExtending*, *EndNormalMode* or *EndFailure*. The *StartExtending* point is triggered when its preconditions are met? the airplane achieves airspeed of less than 200 *knots* and altitude below 2500 *feet*. The *Pilot* then switches the *Handle_Down* causing the *LGCU* to extend the landing gears scenario.

In this exercise of creating UCM scenario models, the Extending Sequence requirements of the LGS are developed and allocated to software items.

The forwards and backwards traceability are established by using two modeling elements in UCM: (1) *UCMmodelElement* has a name and unique identifier; and (2) *Metadata* that is a name-value pair that can be used to attach information to a UCM specification or its model elements.

In the next section, we show how the [*DeploymentSucceeded*] scenario model is transformed into a TDL test scenario.

4.3 Transform UCM scenario models and data model into test scenario in TDL

We explain in details in the following subsections how each element in the TDL specification is developed in the *TDL Builder* process.

A. Develop TDL test objective

In our experimentation, the TDL *Test Objectives* shown in Listing 2 were developed manually by analyzing the sequence and role of UCM objects that reside on the [*DeploymentSucceeded*] scenario and enriched with test requirements.

```
1.  Test Objective TestObj1 {
2.  description: "ensure that when Handle is switched Down, a
                  timer is started. If it times-out 15 seconds later and
                  gears are not locked, a red light is sent";}
3.  Test Objective TestObj2 {
4.  description: "ensure that a 'door locked open light' is received after
                  locking the doors in opened  position"; }
5.  Test Objective TestObj3 {
6.  description: "ensure that an 'amber light' is received when gears
7.                are in transition."; }
8.  Test Objective TestObj4 {
9.  description: "ensure that a 'green light' is received when gears are
                  locked down.";
10. Test Objective TestObj5 {
11. description: "ensure that a 'door locked close light' is received after
                  closing the door "; }
12. }
```

B. Develop TDL data set

In the [*DeploymentSucceeded*] scenario, the *Pilot* and *LGCU* components interact with each other through stimuli and responses. For example, the *Pilot* sends a stimulus to the *LGCU* when executing *Handle_Down responsibility*. The *LGCU* responds by performing internal actions (no interaction) when executing *OpenDoors* and*ClosedDoors responsibilities* and sending responses when stepping into *LockDoorsInOpenedPos*, *AmberON*, *GreenON_AmberOFF*, and *LockDoorsInClosedPos responsibilities*. Table 2 shows the test data for the UCM [DeploymentSucceeded] scenario.

The developed TDL *Data Instances* are grouped in two *Data Set* elements in terms of stimulus and response:

- **GearDeployment:** bounded to *Pilot* messages (Stimulus); and
- **Signal:** bounded to *LGCU* messages (Response).

Listing 3 shows compiled TDL *Data Instances* grouped in two *Data Sets* that are developed from test data in Table 2.

```
1.  Data Set GearDeployment {
2.    instance GearDown;  }
3.  Data Set Signal  {
4.    instance LockOpenedDoor;
5.    instance AmberLight;
6.    instance GreenLight;
7.    instance LockClosedDoor;  }
```

C. Develop TDL Test Configuration

The UCM [*DeploymentSucceeded*] scenario in Fig. 10 is exported, using the UCM *traversal mechanism* [24], to a scenario that contains traversed UCM elements. A snapshot of the exported scenario that highlights the *Test Configuration* is shown in Listing 4. In this exportation, the UCM components Pilot and LGCU are mapped to TDL *Component Instance* objects with a *Gate Instance*. A *Connection* instance is added to indicate that the two *Component Instances* should be connected.

```
1.  <?xml version="1.0" encoding="ISO-8859-1"?>
2.  <tdl:Package xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:tdl="http://www.etsi.org/spec/TDL/20130606"
    name="SuccessfulDeployment">
3.  <comment name="Created" body="April 16, 2016 11:10:12 AM EDT"/>
4.  <comment name="Modified" body="April 16, 2016 11:10:12 AM EDT"/>
5.  <comment name="Author" body="nkesserw"/>
6.  <packagedElements xsi:type="tdl:TestConfiguration">
7.  <componentInstance name="Pilot" type="//@packagedElements.18">
8.  <gateInstance name="gPilot" type="//@packagedElements.6"/>
9.  </componentInstance>
10. <componentInstance name="LGCU" type="//@packagedElements.19">
11. <gateInstance name="gLGCU" type="//@packagedElements.6"/>
12. </componentInstance>
13. <connection name="LGCU_Pilot"
    endPoint="//@packagedElements.0/@componentInstance.2/@gateInstanc
    e.0 //@packagedElements.0/@componentInstance.1/@gateInstance.0"/>
14. </packagedElements>
```

Table 2 Test data for [*DeploymentSucceeded*] scenario

Test data requirements	UCM responsibility (stimulus/response)	TDL data instance
Stimulus to be sent when Pilot switches handle down	*Handle_Down*	*GearDown*
Response to be received when LGCU locks doors in opened position	*LockDoorsInOpenedPos*	*LockOpenedDoor*
Response to be received when LGCU activates gear maneuvering	*AmberON*	*AmberLight*
Response to be received when LGCU locks gears in down position	*GreenON_AmberOFF*	*GreenLight*
Response to be received when LGCU locks doors in closed position	*LockDoorsInClosedPos*	*LockClosedDoor*

Next, a compiled *Test Configuration* element is achieved by parsing the exported scenario to convert the packaged element *tdl:TestConfiguration* into concrete TDL syntax. The *Component Instances* are instantiated to either SUT or Tester, depending on their role. Each *Component Instance* has a gate type to specify the data that can be exchanged, i.e., the *Data Sets* developed earlier. Listing ***5*** shows the TDL *Test Configuration* generated automatically from the exported [*DeploymentSucceeded*] scenario depicted in Listing 4. The *Data Sets GearDeployment* and *Signal* are added manually to the TDL *Test Configuration* (line 1). The two components: *Pilot* and *LGCU* are typed (line 5 and line 8) and connected through newly defined gates (line 11).

```
1.  Gate Type defaultGT accepts GearDeployment, Signal;
2.  Component Type defaultCompType {  gate types
                :defaultGT ; }
3.  Test Configuration TestConfiguration {
4.     //Pilot component
5.    instantiate Pilot as Tester of type
                defaultCompType
6.    having { gate gPilot of type defaultGT ; }
7.     //LGCU component
8.    instantiate LGCU as SUT of type defaultCompType
9.    having{ gate gLGCU of type defaultGT ;  }
10.    //connect the two components through their gates
11.   connect gPilot to gLGCU;  }
```

D. Develop TDL Test Description

The TDL *Data Instances* shown in Listing 3 are used as *Interaction* objects between a Tester and an SUT. The UCM *responsibility* objects along the [*DeploymentSucceeded*] scenario are mapped to the *Action Reference*. UCM scenario Timer Set events are mapped to *TimerStart* objects in TDL. Listing 6 shows a snapshot of the exported [*DeploymentSucceeded*] scenario.

```
1.  <packagedElements xsi:type="tdl:TestDescription"
    name="TestSuccessfulDeployment"
    testConfiguration="//@packagedElements.0">
2.  <behaviour>
3.  <block>
4.  <behaviour xsi:type="tdl:ActionReference" name="Handle_Down"
    action="//@packagedElements.22">
5.  <annotation value="gPilot" key="//@packagedElements.5"/>
6.  </behaviour>
7.  <behaviour xsi:type="tdl:Interaction" name="From Pilot to LGCU"
    source="//@packagedElements.0/@componentInstance.1/@gateInstance.0"
    target="//@packagedElements.0/@componentInstance.2/@gateInstance.0">
8.  <annotation value="Timer_0" key="//@packagedElements.1"/>
9.  <annotation value="If we had a description for this Interaction we could put it
    here." key="//@packagedElements.2"/>
10. </behaviour>
11. <behaviour xsi:type="tdl:TimerStart" name="Timer_0 _Start"
    timer="//@packagedElements.19/@timer.0">
12. <annotation value="gLGCU" key="//@packagedElements.3"/>
13. </behaviour>
14. <behaviour xsi:type="tdl:TimerStop" name="Timer_0 _TimerStop"
    timer="//@packagedElements.19/@timer.0">
15. <annotation value="gLGCU" key="//@packagedElements.3"/>
16. </behaviour>
17. <behaviour xsi:type="tdl:AlternativeBehaviour"
    name="OrFork1291\nisGearsDown">
18. <annotation value="gLGCU" key="//@packagedElements.4"/>
19. </behaviour>
20. <behaviour xsi:type="tdl:ActionReference" name="OpenDoors"
    action="//@packagedElements.23">
21. <annotation value="gLGCU" key="//@packagedElements.5"/>
22. </behaviour>
23. </block>
24. </behaviour>
```

Developing a TDL *Test Description* is automated by parsing the exported [*DeploymentSucceeded*] scenario, extracting the components with their bounded *responsibilities* and mapping them to equivalent TDL objects. Listing 7 shows the TDL *Test Description* that is composed of actions, timers and interactions. As mentioned earlier, the absence of alternative element in the scenario metamodel required manual adjustment of the generated *Test Description* to merge the scenarios that constitute alternate test behavior. This post-processing

is done by adding a TDL element *repeat* "numIteration" attribute (line 7). The element *repeat* iterates over the different alternatives a number of times as determined by the "numIteration" attribute.

```
1.  Test Description TestDescription {  //Test description definition
2.    use configuration : TestConfiguration;  {
3.    perform action Handle_Down on component Pilot with {  PRECONDITION ;  };
4.    gPilot sends instance GearDown to gLGCU with { test objectives :TestObj1;  };
5.    perform action OpenDoors on component LGCU with {  PRECONDITION ;  };
6.    perform action LockDoorsInOpenedPos on component LGCU with {PRECONDITION
      ;} ;
7.    repeat 4 times {  //Iterate over receiving responses, each one is consumed once
8.      alternatively  { // LGCU sends response indicating Door is locked in open position
9.        gLGCU sends instance LockOpenedDoor to gPilot with
10.       { test objectives : TestObj2; };
11.       set verdict to PASS ; }
12.     or { gate gLGCU  is quiet for (7.0 SECOND);
13.       set verdict to FAIL;  }
14.       perform action ReleaseUp_Lock on component LGCU with {  PRECONDITION; };
15.     alternatively  {  // LGCU sends response indicating Gears are in transition
16.       gLGCU sends instance AmberLight to gPilot with { test objectives : TestObj3; };
17.       set verdict to PASS ;   }
18.     or { gate gLGCU is quiet for (7.0 SECOND);
19.       set verdict to FAIL;  }
20.       perform action Lock_DownGears on component LGCU with { PRECONDITION ; };
21.     alternatively  { // LGCU sends response indicating Gears are in locked down
22.       gLGCU sends instance GreenLight to gPilot with { test objectives : TestObj4;  };
23.       set verdict to PASS ; }
24.     or { gate gLGCU is quiet for (7.0 SECOND);
25.       set verdict to FAIL;   }
26.       perform action CloseDoors on component LGCU;
27.       perform action LockDoorsInClosedPos on component LGCU with  {
28.                   PRECONDITION; };
29.     alternatively  { // LGCU sends response indicating Door is locked in close position
30.       gLGCU sends instance LockClosedDoor to gPilot with {test objectives :TestObj5; };
31.       set verdict to PASS ;
32.       perform action ConfirmGearsDown on component Pilot with {PRECONDITION ;};}
33.     or { gate gLGCU is quiet for (7.0 SECOND) ;
34.       set verdict to FAIL;  }
35.     or { gate gLGCU is quiet for (15.0 SECOND);
36.       set verdict to FAIL;  }
37.     }
38.   }
39. }
```

The interactions of the *Test Description* start when a *GearDown* command flows from the *Pilot* gate to the *LGCU* gate (line 4). Immediately afterwards, a timer is started to satisfy the timing constraint of the landing gears' deployment, followed by a second timer to time the action of door opening. Shortly after locking the doors in the opened position, the *LGCU* gate sends the *LockOpenedDoor* sign (line 9) indicating all the doors are locked in the opened position. The *LGCU* releases the *up-lock* and an *AmberLight* status is sent (line 16) indicating the gears are in transition to the full-down position. Another timer is started to time the action of locking the gears in the down position. Gears are locked once they reach the final position, when a *GreenLight* status message is sent from the *LGCU* gate (line 22) indicating full deployment of the landing gears. If the *GreenLight* status message sign is received before any time expiration, a pass verdict is issued and the *Pilot* confirms gears are down and locked (line 32); otherwise, the test fails.

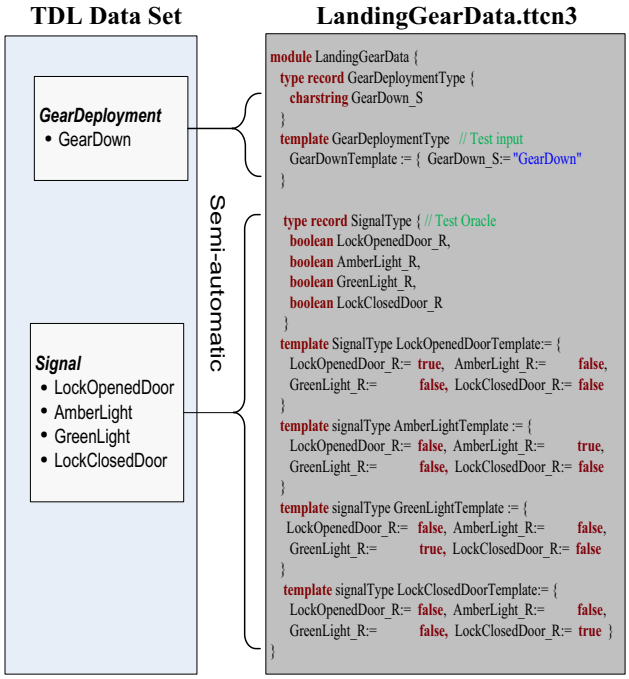


Fig. 11 Mapping abstract TDL data sets to concrete data in TTCN-3

The elements obtained earlier—*Test Objective*, *Data Set*, *precondition* and *Test Configuration*—are used in the *Test Description* to help structure the TDL specification. "Appendix 1 " shows the developed TDL Test Specification. In the next section, we show how to script the obtained TDL specification into executable TPs in TTCN-3.

4.4 Transform TDL specifications to TTCN-3 modules

In the following subsections, we show how the TTCN-3 modules are developed from the TDL specification.

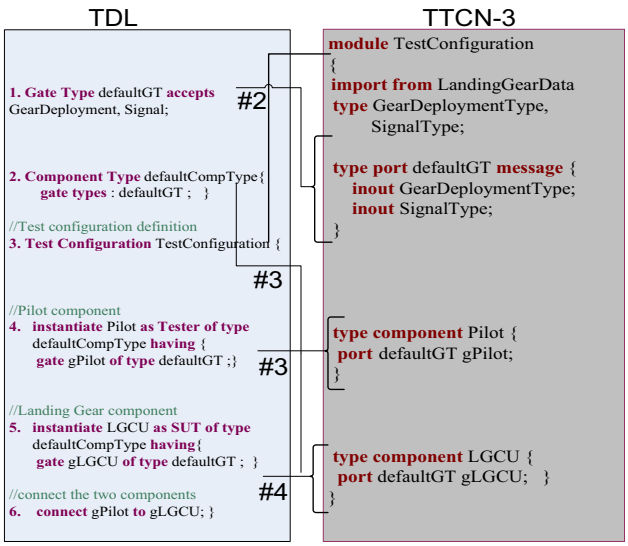
A. Derive the Test Input and Test Oracle modules (TTCN-3) The two *Data Sets; GearDeployment* and *Signal*, defined previously in Listing 3, are parsed with their *instances* to generate records and record fields (variables) in TTCN-3 syntax based on ***Rule# 22*** and ***Rule# 23***. After the TTCN-3 data module is partially generated and test data becomes available, the module is completed with Test Oracle information and typed with concrete TTCN-3 types. Fig. 11 shows the transformation (semiautomatic) between TDL *Data Sets* and TTCN-3 data module.

The *Data instances* developed in the previous section are next mapped to the corresponding TTCN-3 templates by means of TDL data element mappings as shown in Listing 8.

```
1. Use "LandingGearData.ttcn3" as LGearData;
2. Map GearDown to "GearDownTemplate" in LGearData;
3. Map LockOpenedDoor to "LockOpenedDoorTemplate" in
     LGearData;
4. Map AmberLight to "AmberLightTemplate" in LGearData;
5. Map GreenLight to "GreenLightTemplate" in LGearData;
6. Map LockClosedDoor to "LockClosedDoorTemplate" in
     LGearData;
```

B. Derive the Test Configuration module (TTCN-3)

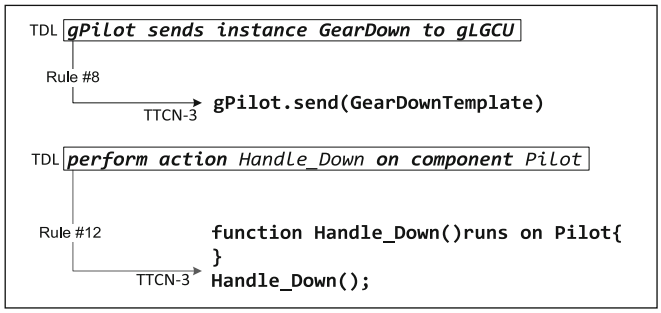
Based on the transformation rules defined in "Appendix 3"; ***Rule# 2***, ***Rule# 3***, and ***Rule# 4***, the transformation of the obtained TDL *Test Configuration* into an equivalent one in TTCN-3 is performed. Listing 9 shows the transformation of one TDL *Test Configuration* into an equivalent one in TTCN-3.



The obtained Test Configuration in TTCN-3 defines test component types and port types, denoted by the keywords **component** and **port**. The communication between the components is achieved via the message-based communication port *gPilot* and *gLGCU,* through which messages of type *GearDeploymentType* and *signalType* are sent and received. The connection between the two components is shown in the Test Description module (developed next) and is expressed with a map function.

C. Derive the Test Description module (TTCN-3)

As mentioned previously, the developed tool maps the TDL elements to TTCN-3 statements. Listing 10 shows an example of transforming two major TDL elements: *action* and *interaction*. The tool parses the *sends instance* statements (interaction) and generates a TTCN-3 message statement. The *action* statement is parsed to generate a function signature and a function call. The obtained function is refined at the TTCN-3 level when applicable.



The transformation of TDL *Test Objectives* cannot be rule-based. However, their semantics can be interpreted manually and reflected in the TTCN-3 Test Description module. "Appendix 2" shows a TTCN-3 Test Description module transformed from the TDL Specification in Listing 7.

Now, the executable TPs is completed by combining the derived modules represented by the three TTCN-3 files: "LandingGearData.ttcn3," "TestConfiguration.ttcn3," and "TestDescription.ttcn3." Listing 11 shows an additional module "DeployLandingGears.ttcn3" to invoke the TP execution.

```
1. module DeployLandingGears {
2.  import from TestDescription testcase _TC;
3.  control { execute(_TC());  }
4. }
```

5 Approach evaluation

The evaluation of the approach is sampled with an industrial product from the private domain, a flight management system (FMS), see Fig. 12.

The FMS test stimuli are key presses and the Test Oracles are screen dumps. Since the FMS functionality was tested using software tests developed manually and determined correctly the FMS behavior, we wanted to evaluate our approach using the same case study in order to assess the efficiency of our approach. We present an empirical evaluation of the approach, based on the results obtained with 3 FMS use cases. We studied the approach efficiency in terms of generating executable TPs and we evaluated the correctness of the generated workflow in two steps:

- **Perform requirement-based test coverage analysis**: we analyzed the TCs, generated from the requirements, to confirm that there is at least one TC for each requirement and all TPs and TCs are traceable to requirements.
- **Perform verdict analysis**: we used a small set of legacy TPs to assess the correctness of the generated TPs. Since the execution of the legacy TPs against SUT reported correctly its behavior and verified that the implementa-

Fig. 12 FMS front panel (photo Esterline CMC Electronics)

tion satisfies the requirements, we used them as an oracle version. We compared our TPs verdicts against the ones emitted by the legacy TPs. The pass verdict indicates correct implementation where the fail verdict indicates an error has been detected.

In this section, we first present the SUT and use cases. Next, we describe the experimental method and give statistics on the workflow generated by our approach.

5.1 The SUT (FMS functionality)

An FMS is typically comprised of the following interrelated functions: navigation, flight planning, trajectory prediction, performance computations, and guidance. It provides the primary navigation, flight planning, optimized route determination and en-route guidance for an aircraft. In order to accomplish these functions, the flight management system must interface with several other avionics systems. A short description of three key functions performed by the FMS and used in the evaluation is given below:

- **Flight Planning:** the flight planning function allows the creation of a flight plan based on the data combinations from a company's route, defined waypoints, navigation database, etc.
- **Lateral Guidance:** This function allows waypoint management via its control display unit interface when an aircraft is configured as a rotor.
- **Navigation:** This function determines the accuracy variable based on the present position, ground speed, and wind speed/wind direction.

5.2 The experimental method

We analyzed the efficiency of the approach by running an experiment aiming to determine whether the approach is efficient to generate TCs that cover the requirements and can be transformed to correct TPs. We consider a TP is correct, after being executed on the FMS, if it reports correctly the behavior of the SUT. Our first step was to select from the legacy software tests a number of TPs that cover the three FMS key functions reported in the previous section. Five legacy TPs that were manually developed, performed on the FMS and reported correctly its behavior covered those functions and therefore were selected. Next, we identified the corresponding requirements of these legacy TPs and grouped them into 3 use cases. The description of each use case is given as follows:

- **Automatic Leg Transitions:** contains 8 functional requirements that specify the automatic leg change using fly-by (turn anticipation) or fly-over (turn over the waypoint).
- **Provide Guidance for a Manual Direct-to Intercept:** contains 7 requirements that specify the operations of the "discontinuity ahead" alter message on the modified route.
- **Predict the Expected Time of Arrival (ETA) with different configurations:** contains 9 requirements that specify the computations to be performed by the FMS for an aircraft to arrive at a certain place.

For each use case, the experimental method we applied consists of:

- Requirement stage: the requirements in the use case were formalized into Cockburn notation and manually mapped to UCM models. We validated the scenario models and checked if they describe correctly all the requirements.
- Test scenario stage: for each possible path in the scenario model, its definition was created and stored as XML file. Using our java-based tool, we transformed the scenario path expressed in XMI format into scenario test expressed in TDL notation. We completed the obtained scenario test (TC) with *Test Objectives* and *Data Instances* elements which are taken mainly from the requirements.
- Test generation stage: based on transformation tool that we implemented with the *Xtext* and *Xtend* framework, we transformed each TC into an executable TP.

Table 3 The executed TPs against the FMS

TP performed on FMS	# of input/output exchanged with FMS	# of verdict per TP
Fly-by procedure	32	10
Fly-over procedure	26	11
Fly-over procedure via DES+SAR	236	129
Manual Direct-to Intercept	116	20
ETA computation	393	168
Total	803	338

- Test execution stage: the resulting TPs that correspond to the selected legacy TPs were executed on the FMS and their test results were recorded.

As a result, 26 TCs and TPs were generated from the 3 use cases. Five TPs were performed on the FMS and their test results were recorded. The selected TPs stimulate the FMS functionality and reflect largely the use cases. Table 3 shows the details about the executed TPs where the description of each TP is given in column 1. Columns 2 and 3 show the number of exchanged messages with the FMS and their verdict, respectively. A total of 803 exchanged messages and 338 test verdicts are performed as shown in the Total row.

5.3 Efficiency in terms of requirement coverage and generating correct TPs

We analyzed the generated TCs to check if they cover the requirements. Table 4 shows that the approach covered all paths in the scenario models effectively. In fact, the approach generated one TC for each scenario path in the scenario model. The total number of the generated TCs successfully covers all possible paths in the UCM model and achieves therefore full scenario and requirement coverage.

The generated TPs were assessed for their correctness by comparing their test results against the legacy TPs. The objective is to have the TPs behavior matches the legacy tests. As mentioned, the legacy tests are used as a golden version to assess the correctness of the generated TPs. Table 5 shows the result of the verdict comparison for each pair of TP. The scenario models that describe the requirements are shown in the first column. Followed by TP description in the second column. The rate of matching verdict with the corresponding legacy test is presented in the third column.

All the verdicts in the *Fly-by procedure* and*Fly-over procedure* TPs matched the corresponding verdicts of the legacy tests. In the remaining TPs, *Fly-over-procedure via DES+SAR, Manual Direct-to Intercept* and *ETA computation*, very few number of verdicts did not match with the corresponding legacy tests. The result in the third column determined with high rate of success the SUT behavior—emitting pass verdict when it is expected and fail verdict in presence of errors.

5.4 Discussion

We applied our approach to a real-life case study FMS at Esterline CMC Electronics. The validation has been achieved by comparing the behavior of the legacy and the generated tests. If they are behavior equivalent, same sequence of test events and verdicts, we can consider them comparable. The verdict of almost all oracle steps in the generated TP matched their corresponding ones in the legacy. In other words, the generated TPs passed and failed in the same steps as the legacy TPs did except a small number of failures in the generated tests. These failures were mostly due to timing issues. The generated tests in TTCN-3 execution have a considerable better performance as the legacy system and the SUT is relatively slow. These cases could be easily detected using the state of the SUT. If the state was the same as for the preceding test event, this indicates that the SUT has not updated its state yet. Here, the responses are not coming spontaneously but instead the test system must query the SUT to obtain the response. Also, some of the failures could indicate that there are alternative behavior in the SUT, something that the legacy

Table 4 The requirement coverage by the generated TCs from UCM model

Use case modeled as scenario	# of scenario path		# of TCs	Requirement coverage rate (%)
	Main	Secondary		
Automatic leg transitions	3	9	12	100
Provide guidance for a Manual Direct-to Intercept	1	7	8	100
Expected time arrival computation	1	5	6	100

Table 5 The matching rate of the executed TPs

Use case modeled as scenario	Executed TP	Verdict matching rate with legacy (%)
Automatic leg transmission	Fly-by procedure	100
	Fly-over procedure	100
	Fly-over procedure via DES+SAR	98
Provide guidance for a Manual Direct-to Intercept	Manual Direct-to Intercept	97
Expected time arrival computation	ETA Computation	98

test system could not handle because it was based on linear sequences of test events.

In conclusion, this study reveals that our approach generated TCs that cover all the described requirements in the scenario models achieving full requirement coverage.

Compared to the legacy testing system, the new approach improves the testing in practice and offers several advantages to the test engineers. We found the following benefits from our new testing practice:

- **Increased test system understanding:** using a model enables to get an overview of the behavior of a system compared to scattered bits and pieces of information.
- **Early Testing:** The test engineers don't need to wait; they describe the requirements in a model, and then push a button to generate the tests.
- **Reduced test effort**: in our model-driven testing, the number of iterations to get correct TPs is reduced. The test development phase is eliminated. The TPs are no longer written by hand or manually corrected, but generated.
- **Traceability:** Documentation can be generated from the model and is thus consistent with the tests. Since TPs are derived from the UCM models where requirements are described, any defect found during the execution of a TP can be traced back to its requirement.
- **Systematic and automation:** with the help of the developed tools, repeated tests are enabled which ensures the robustness of the test results.
- **Reduced human errors**: The fact that the tests are generated from the model and thus consistent with requirements reduces, by definition, the possibility of error in the test suite.

5.4.1 Generalization of the approach

The approach focuses on functional aspects of software and has been applied to two realistic case studies from avionics domain. Additionally, the methodology can be applicable to safety-critical software as it covers timing requirements and provides traceability evidence from requirements to tests. The approach relies on two major elements to improve the testing process:

Modeling: the system requirements (functional) and design are described by high-level visual models and DSL abstracting away technological implementation detail.

Model transformation: the automated model transformations are used to generate tests to reduce the manual work and to simulate high-level models in order to validate the suitability of the modeled system behavior in an early development phase.

Today, the practical realization of model-driven testing benefits from a variety of tools and technologies. Some requirements may not be describable with the UCM notation such as robustness requirements. Such requirements have to be specified by means of other notations or languages. The model transformations are (partially) automated and require little human intervention. The process converts the informal requirements into a formal UCM model. We have used the tool described in [5] that generates individual tests traces, called test scenarios in TDL but as already mentioned, test traces are not always test cases. A good test case comprises alternative behavior both in TDL and in TTCN-3. This part had to be generated manually since no tools exist to perform this task. The hints found in [5] have been tried out and were successful. Thus, the implementation of that part of the translator is future work. However, the translation from TDL to TTCN-3 is relatively straightforward since there is mostly a one to one mapping from TDL to TTCN-3. Only, things such as describing test purposes are not covered and thus have to be translated manually usually as TTCN-3 comments. Overall, our achievement was to show that it is an advantage to build a formal UCM model because everything else down the path can be automatically generated and is either all the way right or all the way wrong. Test automation has the advantage to be systematic when it comes to errors as opposed to manual processes where errors are introduced randomly and are difficult to trace. This automation reduces the required amount of manual work for test development, such that the

testing process is supposed to become less error-prone and more efficient.

5.4.2 Lessons learned

We distill some of the important lessons we have learned in developing and deploying the testing methodology.

The users of the testing methodology should not need to have the functional requirements expressed with use case notation to model them as scenarios. However, requirements presented as a use case facilitated the mapping to UCM models. The model transformation to TDL domain is not fully automatic and requires human intervention to obtain the data elements and to construct the alternatives. The TDL models were a key component of model-driven testing as they have been used as input and output in the model transformation process. The decision to use the TDL notation in the development of tests was successful. TDL narrowed the gap between the described requirements and tests and served as way of communication with non-technical people and as a base to generate concrete tests.

6 Related work

The surveyed papers below propose to generate test cases from UML artifacts which are considered system level and based on use cases and NL.

6.1 Generate test cases from NL

The approach in [30] presents a method (SCENT) to create scenarios from NL and formalize them in state charts. An annotation technique is then used to enrich the state charts with helpful information. A path traversal algorithm is employed in the state charts to determine concrete test cases. The test suite is further enhanced by generating test cases from dependency charts that are modeled from dependencies between scenarios. SCENT requires two different representations of the scenarios, which makes it rather costly in terms of testing effort.

The approach in [31] generates test cases based on NL requirements' specifications using a tool. The tool models the NL requirements into UML activity diagrams to support automated testing. This approach requires using a scenario language [25] that references relevant words from the application with lexicon symbols.

6.2 Generate test cases from use cases

In [12], the authors proposed an approach that links the requirement process with the testing process through a use case model. The approach creates system test cases based on two types of models: (1) UML use case models that describe the system requirements from test designers' point of view; and (2) various forms of MBT. The approach requires additional behavioral modeling such as activity diagram, sequence diagram and class diagram models. The approach focuses on data flows that require manual intervention by test designers to annotate UML diagrams with additional test data such as coverage requirements, constraints and preconditions.

In [27], a model-driven process is proposed to generate automatically both formal models and test cases from the same UML model of the system under verification and validation and model transformation. The approach is applied to a railway control system that features all the characteristics of a complex embedded system. The approach is based on formal methods to reduce the overall assessment effort and to support the validation against both functional and non-functional requirements. However, the formal models are time consuming and expensive to generate and are difficult to be used as a communication mechanism for non-technical personnel.

The authors of [13] proposed an MDT approach for testing applications designed in a model-driven development context (MDE). Their work focuses on the separation of generating test cases and oracles, and the execution of these tests on different target platforms. However, the work considers a specific issue and explicitly addresses the problem of test generation in MDE context.

In [6], the authors propose a methodology TOTEM for system testing to derive system test requirements from early UML artifacts such as use case, class, and sequence diagrams. The authors propose to express the sequential constraints of the use cases with an extended activity diagram that are transformed into a weighted graph. The regular expressions that correspond to use case sequences are extracted from the weighted graph. The derivation of test artifacts from test requirements is delayed till the low level design becomes complete, and when detailed information becomes available regarding application domain and solution domain classes.

The authors of [33] propose to use restricted natural language for the specification of use cases. The use cases are mapped to a formal model (FSM) and test scenarios are generated by traversing the FSM based on coverage criteria. In this approach, there is a substantial overhead for diagram creation and modification of the use case description to the restricted natural language format.

Another important approach to generate test cases from use cases is presented in [28]. The approach generates test cases in two phases. In the first phase, the approach describes system requirements via use case diagram, scenario and contracts. Each use case is enhanced with contracts that are expressed in first order logical expression to specify the preconditions and post-conditions. Next, the enhanced use cases are transformed to test objectives using transition sys-

tem known as Use Case Transition System (UCTS) that can represent all valid sequences of the use case. In the second phase, the test objectives are transformed to test scenarios. Sequence diagrams are attached as additional artifacts to obtain sequences of message calls on the SUT. The approach requires working with various UML diagrams and formal method.

The authors of [5], have explored the automated generation of TDL *Test Descriptions* from requirements expressed as UCM scenario models using the jUCMNav tool. This transformation enables the exploration of model-based testing where the use of TDL models simplifies the generation of tests in various languages such as TTCN-3. The authors determined the basic differences between scenarios and test cases in the handling of alternative paths that result from UCM alternatives. They concluded that the use of scenarios for test case generation is feasible, but requires either a different traversal mechanism with a different scenario metamodel, or post-processing of scenarios to merge those that constitute alternate test behaviors.

In [29], the authors introduced an automatic test generation approach that provides more natural and standardized ways of writing requirements using document templates. These templates are extended to allow include and extension relations between use cases and to include data elements as user-defined types, variables and parameters. The approach uses the use cases templates that capture control flow, state, input and output as source for the generation of formal models. Unfortunately, it only generates non-executable test cases.

Our work aligns with most of the above-mentioned methods (deriving test cases from use cases), but differs by focusing on writing the requirements as abstracts test scenarios and transforming them to executable test cases without worrying about implementation details. Our novel approach provides scenario coverage criteria on requirements, allowing test engineers to prioritize desired requirements to be verified and to start the inspection process earlier. In addition, formalizing test descriptions into abstract test scenarios allows test engineers to focus on the task not on the test data. This particularity motivated us to apply a scenario-based testing methodology. Furthermore, building a testing methodology that is composed of standard-based tools allows organizations to benefit from continuing advancements in modeling and in test scripting tools.

7 Conclusion and future work

The manual transition of NL functional requirements to executable tests requires a considerable amount of effort from the test engineers—especially to ensure acceptable test coverage of the test implementation. Generating tests based on models and model transformations promise to improve the testing process. Model transformations are a key concept for model-driven testing. They are the links between the test artifacts and are used to map models between different domains.

In this paper, an MDT methodology is proposed to generate test artifacts that is manually developed by legacy testing system. The SUT requirements to be tested are described in UCM models which are taken by a model transformation process as input and mapped to abstract test scenarios in TDL. The level of abstraction of the TDL models is more refined by another model transformation to generate executable tests in TTCN-3. The application of the proposed methodology is performed on a real case study.

Full automation (where feasible) of the proposed approach is the aim of our ongoing work in this area, especially the automation of merging linear scenarios (detecting common path up to a UCM branch).

Acknowledgements This research was supported by CRIAQ, Esterline CMC Electronics, Solutions Isoneo and Mitacs-Accelerated Graduate Research Internship Program. Project title: Test Automation with TTCN-3, Grant Numbers: FR05066, FR05067.

Appendix 1: TDL specification developed from UCM [*successful deployment*] scenario model

```
1.  TDLan Specification DeployLandingGearTest {
2.    Verdict PASS; Verdict FAIL;
3.    Action Handle_Down: "when airspeed is less than 200 knots and altitude is less than 2500 feet, the pilot switches handle down and keep it down
                           for 15 seconds, gears starts";
4.    Action OpenDoors: "when doors are locked in closed position, the corresponding cylinder are extended to unlock the doors";
5.    Action LockDoorsInOpenedPos: "lock the doors in opened position";
6.    Action ReleaseUp_Lock: "when gears are locked in up position, the gear cylinders receive hydraulic pressure in order to release the lock that holds
                              the gears";
7.    Action Lock_DownGears: " lock gears when reach full down position";
8.    Action CloseDoors: "when doors are locked in opened position, the corresponding cylinder are extended to unlock the doors";
9.    Action LockDoorsInClosedPos: "lock the doors in closed position";
10.   Action ConfirmGearsDown: "Pilot confirms gears are down and locked";
11.   Annotation PRECONDITION ;
12.   Time Unit SECOND;
13.   Test Objective TestObj1 {
14.   description: "ensure that when Handle is switched Down, a timer is started. If it times-out 15 seconds later and gears are not locked, a red light
                    is sent";}
15.   Test Objective TestObj2 {
16.   description: "ensure that a 'door locked open light' is received after locking the doors in opened position.";}
17.   Test Objective TestObj3 {
18.   description: "ensure that an 'amber light' is received when gears are in transition.";}
19.   Test Objective TestObj4 {
20.   description: "ensure that a 'green light' is received when gears are locked down.";}
21.   Test Objective TestObj5 {
22.   description: "ensure that a 'door locked close light' is received after closing the door.";}
23.   Data Set GearDeployment {
24.   instance GearDown; }
25.   Data Set Signal { instance LockOpenedDoor;  instance AmberLight;  instance GreenLight;  instance LockClosedDoor; }
26.   //Data Instance reference
27.   Use "LandingGearData.ttcn3" as LGearData;
28.   Map GearDown to "GearDownTemplate" in LGearData;
29.   Map LockOpenedDoor to "LockOpenedDoorTemplate" in LGearData;
30.   Map AmberLight to "AmberLightTemplate" in LGearData;
31.   Map GreenLight to "GreenLightTemplate" in LGearData;
32.   Map LockClosedDoor to "LockClosedDoorTemplate" in LGearData;
33.   Gate Type defaultGT accepts GearDeployment, Signal; //Define the gate type and the exchanged data set
34.   Component Type defaultCompType {  gate types :defaultGT ; }
35.   Test Configuration TestConfiguration  { // Pilot and LGCU
36.     instantiate Pilot as Tester of type defaultCompType having { gate gPilot of type defaultGT ;  }
37.     instantiate LGCU as SUT of type  defaultCompType having{ gate gLGCU of type  defaultGT ;   }
38.    connect gPilot to gLGCU;  }   //connect the two components through their gates
39.   Test Description TestDescription {  //Test description definition
40.     use configuration : TestConfiguration;  {
41.     perform action Handle_Down on component Pilot with {  PRECONDITION ;  };
42.     gPilot sends instance GearDown to gLGCU with { test objectives :TestObj1;   };
43.     perform action OpenDoors on component LGCU with {  PRECONDITION ;  };
44.     perform action LockDoorsInOpenedPos on component LGCU with {  PRECONDITION ;  };
45.     repeat 4 times {   //Iterate over receiving responses, each one is consumed once
46.       alternatively  { // LGCU sends response indicating Door is locked in opened position
47.          gLGCU sends instance LockOpenedDoor to gPilot with { test objectives : TestObj2; }; set verdict to PASS ; }
48.       or { gate gLGCU  is quiet for (7.0 SECOND); set verdict to FAIL;  }
49.          perform action ReleaseUp_Lock on component LGCU with {  PRECONDITION ;  };
50.       alternatively  { // LGCU sends response indicating Gears are in transition
51.          gLGCU sends instance AmberLight to gPilot with { test objectives : TestObj3; }; set verdict to PASS ;    }
52.       or { gate gLGCU is quiet for (7.0 SECOND); set verdict to FAIL;  }
53.          perform action Lock_DownGears on component LGCU with {  PRECONDITION ;  };
54.       alternatively  { // LGCU sends response indicating Gears are locked down
55.          gLGCU sends instance GreenLight to gPilot with { test objectives : TestObj4;  };  set verdict to PASS ; }
56.       or { gate gLGCU is quiet for (7.0 SECOND); set verdict to FAIL;    }
57.          perform action CloseDoors on component LGCU;
58.          perform action LockDoorsInClosedPos on component LGCU with {  PRECONDITION ;  };
59.       alternatively  {  // LGCU sends response indicating Door is locked in closed position
60.         gLGCU sends instance LockClosedDoor to gPilot with { test objectives : TestObj5; }; set verdict to PASS ;
61.         perform action ConfirmGearsDown on component Pilot with {  PRECONDITION ;  }; }
62.       or { gate gLGCU is quiet for (7.0 SECOND) ; set verdict to FAIL;  }
63.       or { gate gLGCU is quiet for (15.0 SECOND);
64.          set verdict to FAIL;  }
65.     }
66.    }
67.  }
68.  }
```

Appendix 2: The Test Description module in TTCN-3 mapped from TDL Test Description

```
module TestDescription {
  import from TestConfiguration all;
  import from LandingGearData all;
  testcase _TC () runs on Pilot {
  map (mtc:gPilot, system:gLGCU);
  timer deploymentTime; timer lockDoorOpenedTime;  timer gearsManeouvringTime;
  timer gearLockedDownTime;  timer lockDoorClosedTime;
  Handle_Down();  // function call
  gPilot.send(GearDownTemplate);
  deploymentTime.start(15.0);
  OpenDoors();  // function call
  LockDoorsInOpenedPos ();
  lockDoorOpenedTime.start(7.0);
  alt {
  [] gPilot.receive(LockOpenedDoorTemplate) {
     lockDoorOpenedTime.stop;
     setverdict(pass);
    ReleaseUp_Lock();  // function call
    gearsManeouvringTime.start(7.0);
    repeat  } // restart the alt
  [] lockDoorOpenedTime.timeout {
    setverdict(fail) }
  [] gPilot.receive(AmberLightTemplate) {
    gearsManeouvringTime.stop;
    setverdict(pass);
    Lock_DownGears();  // function call
    gearLockedDownTime.start(7.0);
    repeat   } // restart the alt
  [] gearsManeouvringTime.timeout {
    setverdict(fail) }
  [] gPilot.receive(GreenLightTemplate) {
    gearLockedDownTime.stop;
    setverdict(pass);
    CloseDoors();  // function call
    LockDoorsInClosedPos();
   lockDoorClosedTime.start(7.0);
   repeat } // restart the alt
  [] gearLockedDownTime.timeout {
    setverdict(fail) }
  [] gPilot.receive(LockClosedDoorTemplate) {
    lockDoorClosedTime.stop;
   deploymentTime.stop;
   setverdict(pass);
   ConfirmGearsDown(); } // function call
  [] lockDoorClosedTime.timeout {
    setverdict(fail) }
  [] deploymentTime.timeout {
    setverdict(fail) }  }
  unmap (mtc:gPilot, system:gLGCU);  }  }
  function Handle_Down () runs on Pilot {  }
  function OpenDoors () runs on Pilot {  }
  function LockDoorsInOpenedPos () runs on Pilot {  }
  function ReleaseUp_Lock () runs on Pilot {  }
  function Lock_DownGears () runs on Pilot {  }
  function CloseDoors () runs on Pilot {  }
  function LockDoorsInClosedPos () runs on Pilot {  }
  function ConfirmGearsDown () runs on Pilot {  }
}
```

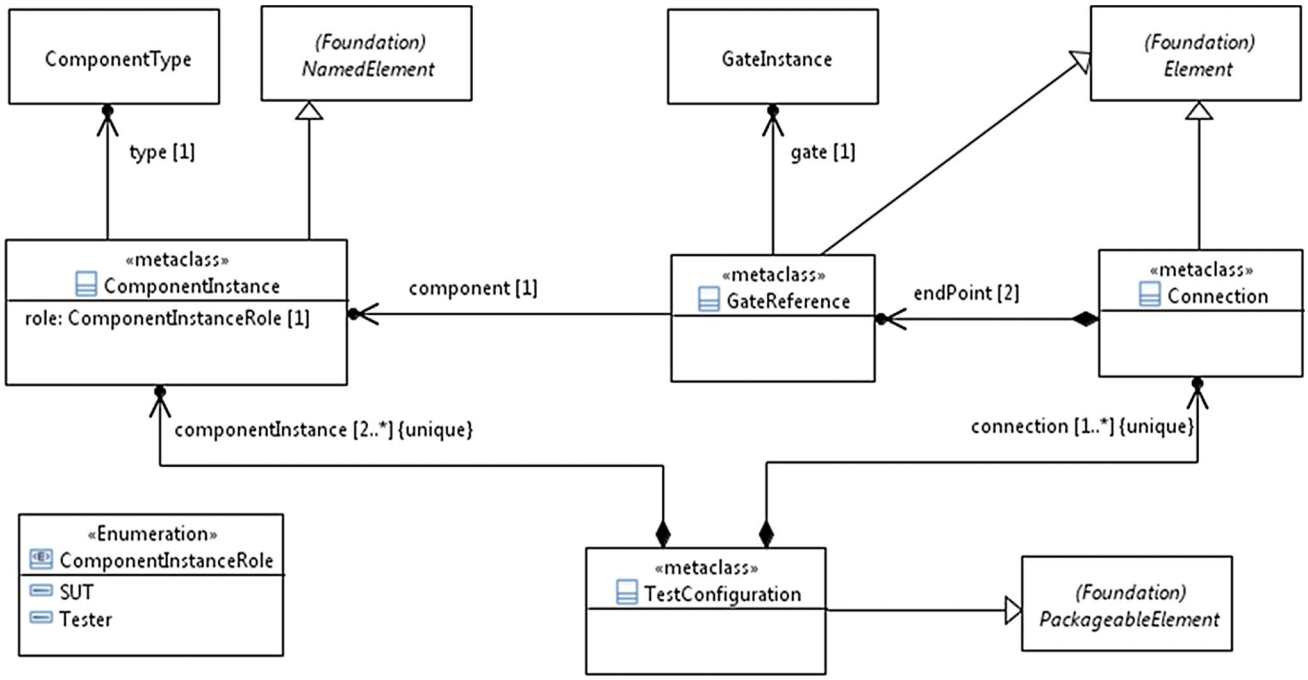
Appendix 3: Transformation rules from TDL to TTCN-3 implemented in Xtend

	TDL metamodel elements (abstract syntax)	Our TDL concrete syntax	Equivalent TTCN-3 statements	Description
Rule# 1	TestConfiguration	Test Configuration < tc_name >	module < tc_name > { }	Map to a module statement with the name < td_name >
Rule# 2	GateType	Gate Type < gt_name > accepts dataOut, dataIn;	type port < gt_name > message { inout dataOut; inout dataIn;}	Map to a port-type statement (message-based) that declares concrete data to be exchanged over the port
Rule# 3	ComponentType	Component Type < ct_name > { gate types : < gt_name > instantiate < comp_name1 > as Tester of type < ct_name > having { gate < g_name1 > of type < gt_name >; }	type component comp_name1{ port < gt_name > < g_name1 >;}	Map to a component-type statement and associate a port to it. The port is not a system port
Rule# 4	ComponentType	Component Type < ct_name > { gate types : < gt_name > instantiate < comp_name2 > as SUT of type < ct_name > having { gate < g_name2 > of type < gt_name >; }	type component comp_name2{ port < gt_name > < g_name2 >;}	Map to a component-type statement and associate a port of the test system interface to it
Rule# 5	Connection	connect < g_name1 > to < g_name 2 >	map (mtc: < g_name1 >, system: < g_name2 >)	Map to a map statement where a test component port is mapped to a test system interface port
Rule# 6	TestDescription	Test Description (< dataproxy) < td_name > { use configuration: < tc_name >; { }}	module < td_name > {import from < dataproxy > all; import from < tc_name > all; testcase _TC() runs on comp_name1 {}}	Map to a module statement with the name < td_name >. The TDL < DataProxy > element passed as a formal parameter (optional) is mapped to an import statement of the < DataProxy > to be used in the module. The TDL property Test Configuration associated with the "TestDescription" is mapped to an import statement of the Test Configuration module A test case definition is added
Rule# 7	AlternativeBehaviour	alternatively { }	alt {}	Map to an alt statement
Rule# 8	Interaction	< comp_name1 > sends instance < instance_outX > to < comp_name2 >	< comp_name1 > .send(< instance_outX >)	Map to a send statement that sends a stimulus message
		< comp_name2 > sends instance < instance_Inx > to < comp_name2 >	< comp_name1 > .receive(< instance_InX >)	Map to a receive statement that receives a response when the sending source is an SUT component
Rule# 9	VerdictType	Verdict < verdict_value >	verdicttype	< verdict_value > contains the following values: {inconclusive, pass, fail}. No mapping is necessary since these values exist in TTCN-3

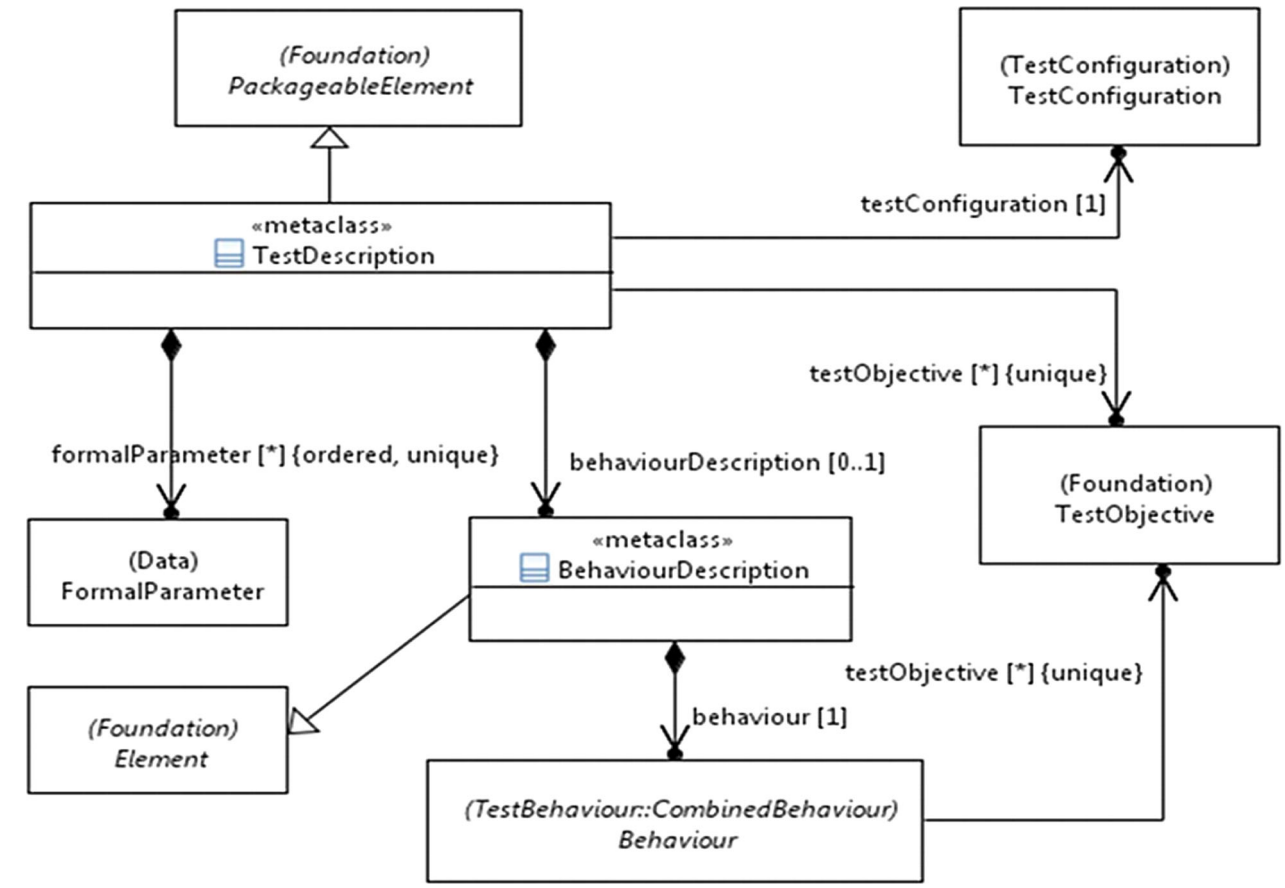
	TDL metamodel elements (abstract syntax)	Our TDL concrete syntax	Equivalent TTCN-3 statements	Description
Rule# 10	TimeUnit	Time Unit < time_unit >	N/A	< time_unit > contains the following values: {tick, nanosecond, microsecond, millisecond, second, minute, hour}. No mapping is necessary; a float value is used to represent the time in seconds
Rule# 11	VerdictAssignment	set verdict to < verdict_value >	setverdict (< verdict_value >)	Map to a setverdict statement
Rule# 12	Action	perform action < action_name >	function < action_name >() runs on < g_name1 >{ } < action_name (); >	Map to a function signature and to a function call. The function body is refined later if applicable
Rule# 13	Stop	stop	stop	Map to a stop statement within an alt statement
Rule# 14	Break	break	break	Map to a break statement within an alt statement
Rule# 15	Timer	timer < timer_name >	timer < timer_name >	Map to a timer definition statement
Rule# 16	TimerStart	start < timer_name > for (time_unit)	< timer_name > .start(time_unit);	Map to a start statement
Rule# 17	TimerStop	stop < timer_name >	< timer_name >.stop;	Map to a stop statement
Rule# 18	TimeOut	< timer_name > times out	< timer_name >.timeout;	Map to a timeout statement
Rule# 19	Quiescence/Wait	is quite for (time_unit) waits for (time_unit)	timer < timer_name > < timer_name >.start(time_unit); < timer_name >.timeout	Map to a timer definition statement, a start statement and to a timeout statement
Rule# 20	InterruptBehaviour	interrupt	stop	Map to stop statement
Rule# 21	BoundedLoopBehaviour	repeat < number > times	repeat	Map to a repeat statement. The repeat is used as the last statement in the alt behavior. It should be used once for each possible alternative
Rule# 22	DataSet	Data Set < DataSet_name > { }	type record < DataSet_nameType > { }	Map Data Set to record type using DataSet_name and prefixed with "Type"
Rule# 23	DataInstance	instance < instance_name >;	[< instance_name_S >;] [< instance_name_R >;]	Map instance to a variable, using instance_name and prefixed either with "_S" for stimulus or with "_R" for response

Appendix 4

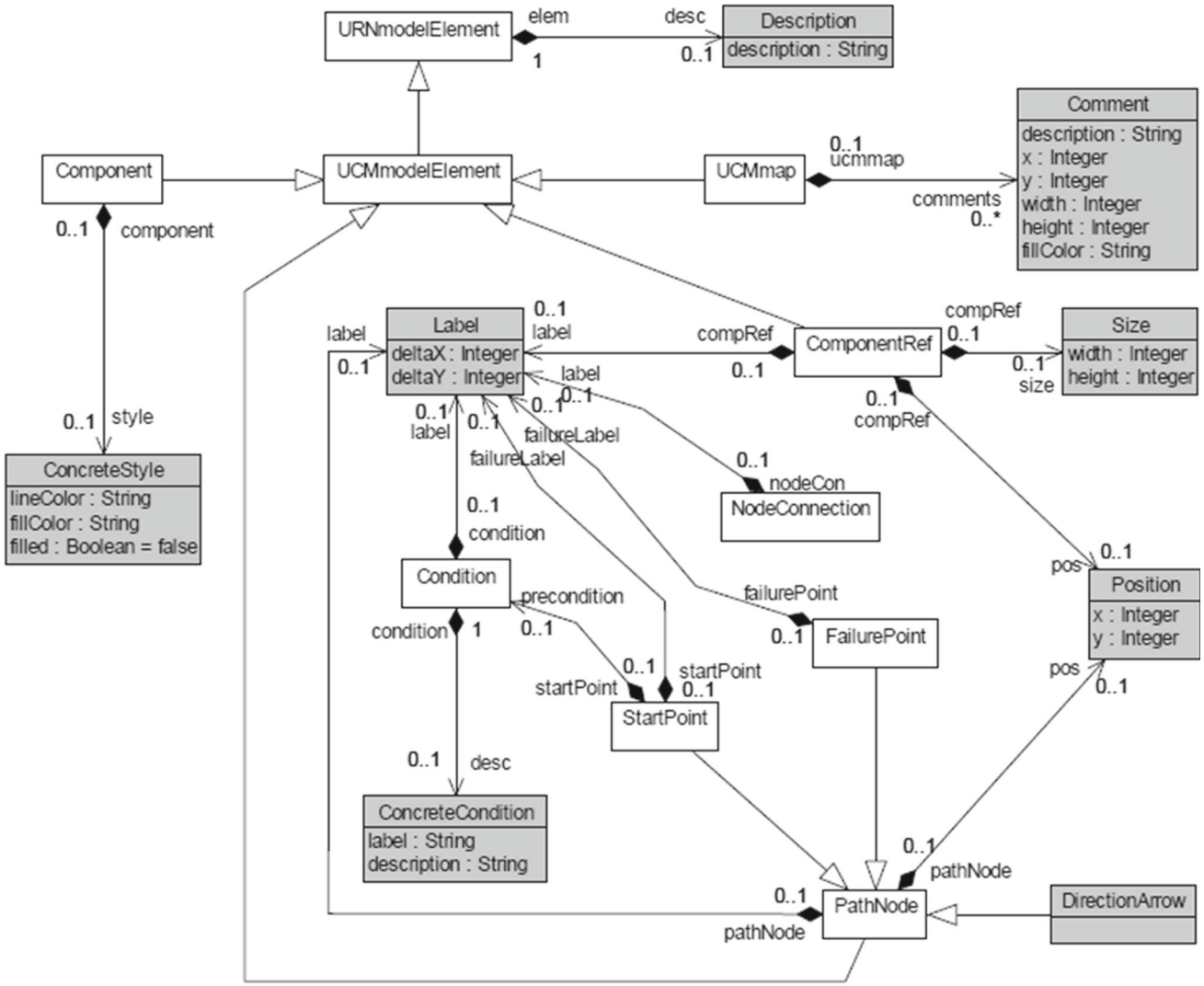
TDL Test Configuration metamodel



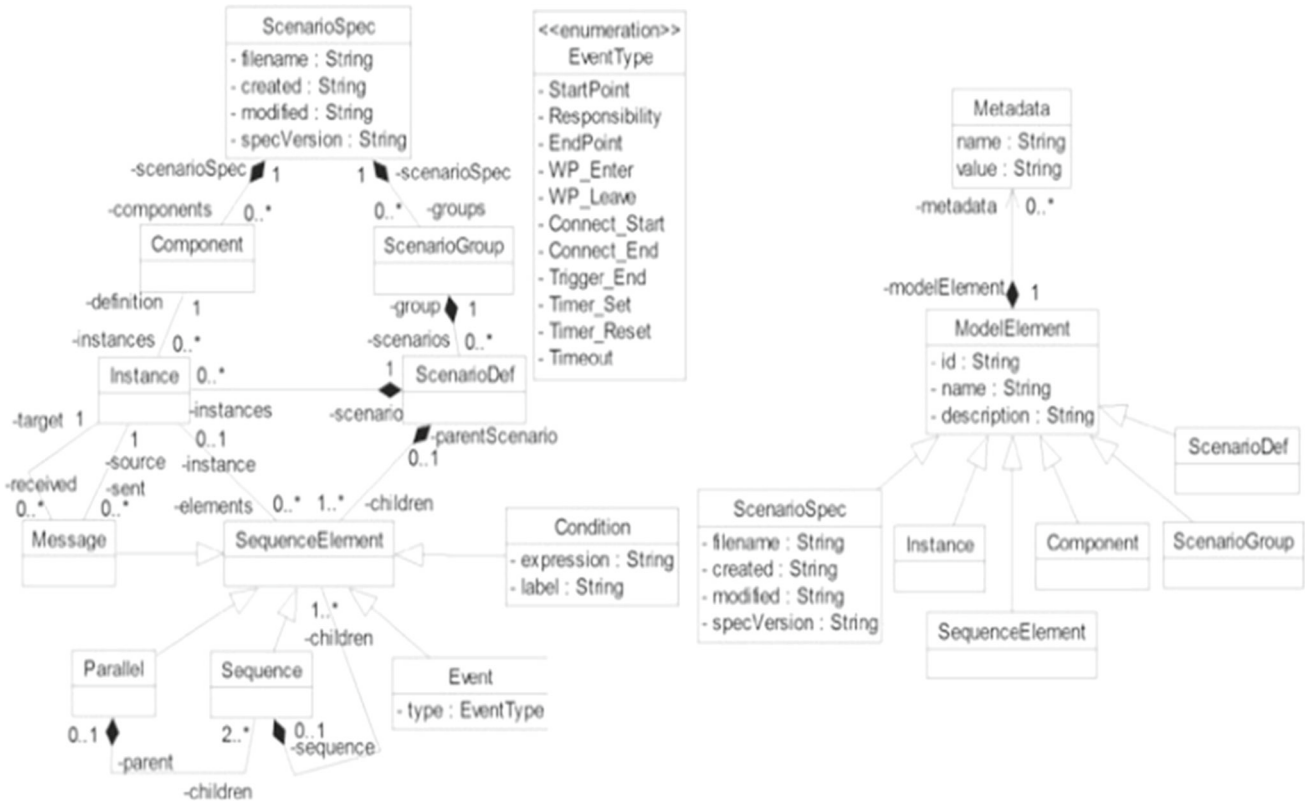
The Test Description metamodel



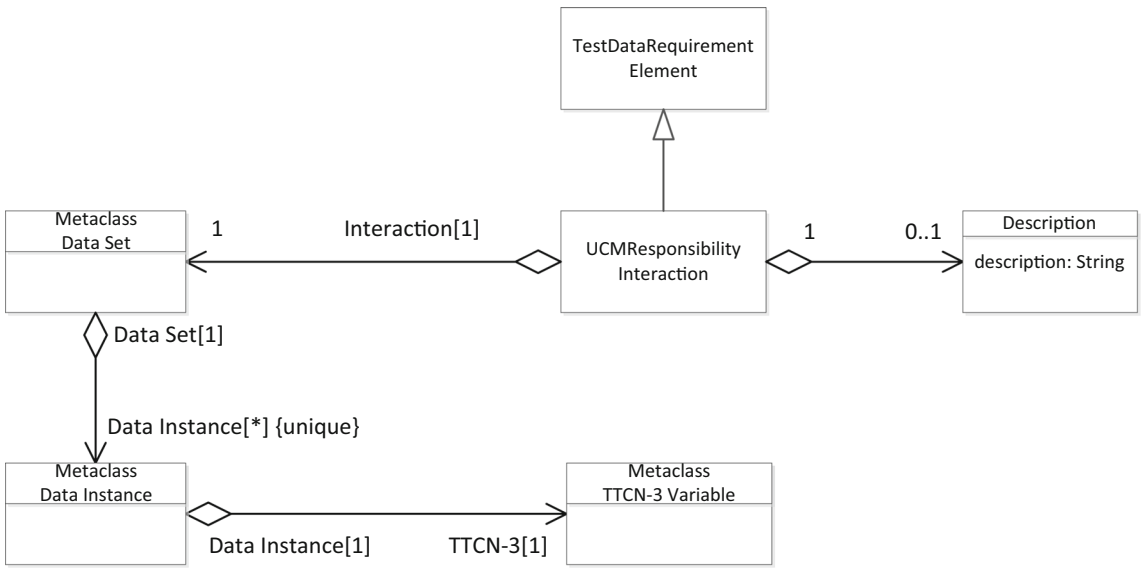
Appendix 5: The concrete metamodel of the UCM notation



Appendix 6: The scenario metamodel



The data metamodel



References

1. Adolph, S., Cockburn, A., Bramble, P.: Patterns for Effective Use Cases. Addison-Wesley Longman Publishing Co., Inc., Boston (2002)
2. Baker, P., Dai, Z.R., Grabowski, J., Schieferdecker, I., Williams, C.: Model-Driven Testing: Using the UML Testing Profile. Springer, Berlin (2007). ISBN 9783540725626
3. Bertolino, A., Fantechi, A., Gnesi, S., Lami, G.: Product line use cases: Scenario-based specification and testing of requirements. In: Software Product Lines, pp. 425–445. Springer, Berlin Heidelberg (2006)
4. Boniol, F., Wiels, V.: The landing gear system case study. In: ABZ 2014: The Landing Gear Case Study, pp. 1–18. Springer (2014)
5. Boulet, P., Amyot, D., Stepien, B.: Towards the generation of tests in the test description language from use case map models. In: SDL 2015: Model-Driven Engineering for Smart Cities, pp. 193–201. Springer (2015)
6. Briand, L., Labiche, Y.: A UML-based approach to system testing. Softw. Syst. Model. **1**(1), 10–42 (2002)
7. Buhr, R.J.A.: Use case maps as architectural entities for complex systems. IEEE Trans. Softw. Eng. **24**(12), 1131–1155 (1998)
8. DO-178A Software Considerations in Airborne Systems and Equipment Certification, Document Number: DO-178A, Issue Date: 3/22/1985, Committee: SC-152, Category: Software
9. DO-178C: Available from RTCA at www.rtca.org
10. Dvorak, D.: NASA study on Flight Software Complexity. NASA office of chief engineer (2009)
11. Elberzhager, F., Rosbach, A., Münch, J., Eschbach, R.: Reducing test effort: a systematic mapping study on existing approaches. Inf. Softw. Technol. **54**(10), 1092–1106 (2012)
12. Hasling, B., Goetz, H., Beetz, K.: Model based testing of system requirements using UML use case models. In: 2008 1st International Conference on Software Testing, Verification, and Validation, pp. 367–376. IEEE (2008, April)
13. Heckel, R., Lohmann, M.: Towards model-driven testing. Electron. Notes Theor. Comput. Sci. **82**(6), 33–43 (2003). ISBN 1571-0661
14. http://jucmnav.softwareengineering.ca/ucm/bin/view/ProjetSEG/WebHome
15. http://www.etsi.org/deliver/etsi_es/203100_203199/20311901/01.03.01_60/es_20311901v010301p.pdf. http://www.etsi.org/technologies-clusters/technologies/test-description-language
16. http://www.ttcn-3.org/index.php/downloads/standards
17. http://xtext.com/
18. http://www.rtca.org/
19. http://jucmnav.softwareengineering.ca/ucm/pub/UCM/VirLibTutorial99/UCMquickRef.pdf
20. http://www.etsi.org/deliver/etsi_es/201800_201899/20187301/04.08.01_60/es_20187301v040801p.pdf
21. https://standards.ieee.org/findstds/standard/830-1998.html
22. Hovsepyan, A., Van Landuyt, D., Michiels, S., Joosen, W., Rangel, G., Fernandez Briones, J., Depauw, J.: Model-driven software development of safety-critical avionics systems: an experience report. In: 1st International Workshop on Model-Driven Development Processes and Practices co-located with ACM/IEEE 17th International Conference on Model Driven Engineering Languages & Systems (MoDELS 2014), vol. 1249 (2014, September)
23. ITU-T Z.151: http://www.itu.int/rec/T-REC-Z.151/en
24. Kealey, J., Amyot, D.: Enhanced use case map traversal semantics. In: Gaudin, E., Najm, E., Reed, R. (eds.) SDL 2007. LNCS, vol. **4745**, pp. 133–149. Springer, Heidelberg (2007)
25. Leite, J.C.S.P., Hadad, G., Doorn, J., Kaplan, G.: A scenario construction process. Requir. Eng. J. **5**(1), 38–61 (2000)
26. Makedonski, P., Adamis, G., Käärik, M., Ulrich, A., Wendland, M.-F., Wiles, A.: Bringing TDL to users: a hands-on tutorial. In: User Conference on Advanced Automated Testing (UCAAT 2014), Munich
27. Marrone, S., Flammini, F., Mazzocca, N., Nardone, R., Vittorini, V.: Towards model-driven V&V assessment of railway control systems. Int. J. Softw. Tools Technol. Transf. **16**(6), 669–683 (2014)
28. Nebut, C., Fleurey, F., Le Traon, Y., Jezequel, J.M.: Automatic test generation: a use case driven approach. IEEE Trans. Softw. Eng. **32**(3), 140–155 (2006)
29. Nogueira, S., Sampaio, A., Mota, A.: Test generation from state based use case models. Formal Asp. Comput. **26**(3), 441–490 (2014)
30. Ryser, J., Glinz, M.: A scenario-based approach to validating and testing software systems using statecharts. In: Proceedings of 12th International Conference on Software and Systems Engineering and Their Applications (1999, December)
31. Sarmiento, E., Sampaio do Prado Leite, J. C., Almentero, E.: C&L: generating model based test cases from natural language requirements descriptions. In: 2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET), pp. 32–38. IEEE (2014, August)
32. Schatz, Bernhard.: 10 years model-driven—what did we achieve?. In: Proceedings of the 2011 Second Eastern European Regional Conference on the Engineering of Computer Based Systems (ECBS-EERC '11). IEEE Computer Society, Washington, DC, USA, 1-. (2011). doi:10.1109/ECBS-EERC.2011.42
33. Somé, S. S., Cheng, X.: An approach for supporting system-level test scenarios generation from textual use cases. In: Proceedings of the 2008 ACM Symposium on Applied computing, pp. 724–729. ACM (2008, March)
34. Ulrich, A., Jell, S., Votintseva, A., Kull, A.: The ETSI Test Description Language TDL and its application. In: 2014 2nd International Conference on Model-Driven Engineering and Software Development (MODELSWARD), pp. 601–608. IEEE (2014, January)
35. What is the Benefit of a Model-Based Design of Embedded Software Systems in the Car Industry?Manfred Broy (Technical University Munich, Germany), Sascha Kirstan (Altran Technologies, Germany), Helmut Krcmar (Technical University Munich, Germany) and Bernhard Schätz (Technical University Munich, Germany). doi:10.4018/978-1-61350-438-3.ch013
36. Zhang, M., Yue, T., Ali, S., Zhang, H., Wu, J.: A systematic approach to automatically derive test cases from use cases specified in restricted natural languages. In: Proceedings of the 8th International Conference on System Analysis and Modeling: Models and Reusability (SAM'14) (2014)

Nader Kesserwan is a Ph.D. student in Information and Systems Engineering at Concordia University, Montreal, Canada. He received his Master degree in Computer Science from McGill University, Montreal, Canada in 1999. He has been a lecturer at several universities from 2000 to 2012 and a senior Quality Assurance Analyst since 2014. His research interests include model-based testing, model-driven testing, test automation, wireless network and software engineering.

Rachida Dssouli is professor and founding Director of Concordia Institute for Information Systems Engineering (CIISE), Concordia University. She received the Doctorat d'Université degree in Computer Science from the Université Paul-Sabatier of Toulouse, France, in 1981, and the Ph.D. degree in Computer Science in 1987, from the University of Montréal, Canada. She has been a professor at the Université Mohamed 1er, Oujda, Morocco, from 1981 to 1989, assistant professor at the Université de Sherbrooke, from 1989 to 1991, and full professor at the Université de Montréal until May 2001. Her research area is in communication software engineering, requirements engineering and service computing. Ongoing projects include incremental specification and analysis of reactive systems based on scenario language, service computing and service composition, multimedia applications, conformance testing, design for testability , conformance testing based on FSM /EFSM and Timed automata.

Jamal Bentahar received his Ph.D. degree in computer science and software engineering from Laval University, Canada, in 2005 and his Master in Software Engineering from École Nationale Supérieure d'Informatique et d'Analyse des Systèmes, Morocco in 1999. He is a full professor with Concordia Institute for Information Systems Engineering, Faculty of Engineering and Computer Science, Concordia University, Canada. From 2005 to 2006, he was a postdoctoral fellow with Laval University, and then an NSERC PDF fellow at Simon Fraser University, Canada. His research interests include services computing, applied game theory, computational logics, model checking, multi-agent systems, and software engineering.

Bernard Stepien holds a Master degree from the University of Montpellier in France. Subsequently, he carried out research in Transportation Science with the Montreal Transportation Commission and worked as an economist for Bell Canada. He has been a private consultant in computer applications since 1975. He has been active in research on Formal Description techniques with the University of Ottawa since 1985. Currently he is involved in various aspects of communication protocols software with the Canadian Government (Department of Industry, Atomic Energy Control Board), Bell Canada and Nortel.

Pierre Labrèche is a software engineering manager at Esterline CMC Electronics; his field of expertise includes embedded systems, software processes, and safety-critical systems engineering. Since his graduation in Electrical Engineering from École Polytechnique de Montréal in 1977, he has gathered experience in avionics, communications, and automation. He started in avionics with navigation and landing systems and is currently active in integrated cockpits. Under Pierre's direction, CMC reached SEI Capability and Maturity Model Level 3. He holds a Six Sigma black belt certification. He teaches avionics and computer courses, parttime. Current interests in safety-critical systems include architecture, model-based development, large systems design and verification, test technologies, and object-oriented technologies.