TRANSFORMING SOFTWARE REQUIREMENTS INTO TEST CASES VIA MODEL TRANSFORMATION

Nader Kesserwan[1], Jameela Al-Jaroodi[1], Nader Mohamed[2] and Imad Jawhar[3]

[1]Department of Engineering, Robert Morris University, Pittsburgh, USA
[2]Department of Computing and Engineering Technology, Pennsylvania Western University, California, Pennsylvania, USA.
[3]Faculy of Engineering, AlMaaref University, Beirut, Lebanon

ABSTRACT

Executable test cases originate at the onset of testing as abstract requirements that represent system behavior. Their manual development is time-consuming, susceptible to errors, and expensive. Translating system requirements into behavioral models and then transforming them into a scripting language has the potential to automate their conversion into executable tests. Ideally, an effective testing process should start as early as possible, refine the use cases with ample details, and facilitate the creation of test cases. We propose a methodology that enables automation in converting functional requirements into executable test cases via model transformation. The proposed testing process starts with capturing system behavior in the form of visual use cases, using a domain-specific language, defining transformation rules, and ultimately transforming the use cases into executable tests.

KEYWORDS

Model-Driven Testing, Transformation Rules, Model Transformation, TDL, UCM & TTCN-3

1. INTRODUCTION

The complexity of software development is on the rise in the modern era, leading to a surge in the need for software verification. Implementing an inappropriate testing methodology could undermine system safety. This is especially true in the avionics industry, where there has been a significant increase in safety-critical software, whether for military or civilian use. One of the key challenges faced by testing engineers is time constraints, which often limit the opportunity for detailed calculations. During the software development phase, the manual generation of test artifacts continues to be a significant cost driver, accounting for over 50% of the total development effort [1]. Automating the testing process can enhance software quality and ensure the reliability of the test results, thereby reducing liability costs and human effort. In the realm of software engineering, there's a growing trend of using scenarios to gather, document, and validate requirements [2]. Scenarios, which depict a series of behavior-related actions, can simplify an application's complexity, thereby facilitating better comprehension and prioritization of the required scenario. System requirements, including functional and operational ones, are encapsulated in these scenarios and leveraged to define test cases (TCs).

A scenario offers insights into the practical implementation of a system's behavior. Since the collective scenarios of a system embody its behavior and functional domain, they ensure statement and decision coverage when the system is segmented into test scenarios. When use cases are employed to model requirements, a scenario can follow a specific route through the model to instantiate a use case [3], [4]. As a result, when a use case is chosen for testing, all of its

DOI: 10.5121/ijsea.2023.14401

potential test scenarios can be utilized to meet a branch coverage criterion. The execution of these test scenarios on the system under test (SUT) can evaluate the comprehensiveness of the path coverage criterion. Hence, these path coverage metrics cater to the safety requirements and assist test engineers in identifying redundant or missing test scenarios.

Our aim is to investigate an alternative testing methodology that facilitates test automation, commences with the representation of requirements, formalizes test descriptions irrespective of the test scripting language, and targets a testing language. Automation in the testing process has the potential to decrease testing effort, minimize human errors, initiate testing earlier, and support the auto-generation of testing artifacts. We were inspired to develop a testing methodology that can employ scenario-based notations to assist in the derivation of the TCs and utilize standard notations to capture and test functional requirements.

The structure of this paper is as follows: Section 2 reviews relevant literature; Section 3 provides necessary background information. Section 4 introduces the proposed approach, which is then evaluated in Section 5. Finally, Section 6 concludes the paper and outlines directions for future work.

2. RELATED WORK

Numerous techniques have been suggested in academic literature for generating test cases from use cases expressed in Natural Language (NL) and UML diagrams. A few of these methods, relevant to this research, are highlighted in the following papers.

Nogueira et al. [5] suggested a method that standardizes the capture of requirements using document templates. This automatic test generation approach extends the templates to allow inclusion and extension relations between use cases. Moreover, the technique allows the inclusion of data elements as parameters, user-defined types, and variables. However, this approach does not generate executable test cases, as the templates that capture control flow, state, input, and output are solely used for creating formal models.

Sarmiento et al. [6] devised a tool that models NL requirements using UML activity diagrams. Their method generates test cases from the activity diagrams to facilitate automated testing. However, the technique has a drawback: it necessitates the use of lexicon symbols to reference pertinent words.

Somé et al. [7] put forward an approach that specifies use cases with restricted NL, which are later mapped to Finite State Machine (FSM) models. A traversal algorithm navigates through the FSM models and generates test scenarios based on a coverage criterion. However, this approach requires modifying the use cases to the restricted NL and creating FSM diagrams, which are considered overhead.

Heckel et al. [8] suggested a Model-Driven Testing (MDT) approach that decouples the generation of test cases from their execution on various target platforms. This strategy for testing applications is crafted within a model-driven development context. However, its application is confined to generating test cases in a model-driven context.

Ryser et al. [9] introduced the SCENT method for developing scenarios based on NL requirements. These scenarios are formalized into state charts, supplemented with additional information, and then flattened by a traversal algorithm to determine the test cases. Further tests are modeled in dependency charts and generated from the interdependencies between scenarios, which increases the testing effort.

Kesserwan et al. [10] outlines a method to reverse engineer legacy software tests to align with a model-driven testing methodology. The traditional test procedures are initially converted into the TTCN-3 language. Following this conversion, these procedures are abstracted into test cases presented in the TDL format.

describe an approach to reverse engineer legacy software tests to a model-driven testing methodology. The legacy test procedures are translated to the TTCN-3 (Testing and Test Control Notation) language and then abstracted to test cases in TDL (Test Description Language) format. The latter is a formal language for expressing test cases.

Numerous methodologies, such as those in Maurer et al. [11], Leonhardt et al. [12], and Larisa et al. [13], leverage implicit relationships to facilitate test generation, execution, and evaluation, while others, like in Labiche et al. [14], utilize implicit relationships for regression testing. Additional methods use explicit relationships to aid in test generation as in Bertolino et al. [15], test execution and evaluation as in Nachmanson et al. [16], or coverage analysis.

Like most of the methods mentioned above, our work also derives test cases from use cases. However, it distinguishes itself by formalizing the requirements as abstract test scenarios and converting them into executable test cases. This segregation of test specification from test implementation provides greater flexibility for test deployment and allows test engineers to concentrate on the test objectives.

Our approach offers scenario coverage criteria and enables prioritizing desired requirements to be tested early in the process.

3. BACKGROUND

In the realm of software development, numerous strategies have been devised to modernize testing procedures. One such technique is model-driven testing, which automates the creation and transformation of models based on transformation rules defined via mappings between metamodel elements. In this model transformation engineering, models at various levels of abstraction facilitate generalization and automated development.

Another method employed to update testing processes is specification-based testing (SBT). This approach has been utilized in the testing of intricate software systems. SBT verifies that the software or system meets the requirements and delivers value. Consequently, test engineers benefit from the accuracy and the detailed depiction of system requirements provided by the SBT approach. The tests are developed from the requirements' context and designed from the user's perspective rather than the designer's.

There are several modeling languages available to articulate functional requirements and numerous languages that can be used to specify or describe test cases (TCs). These TCs can be manually developed or automatically derived from behavior models. Some of the modeling notations used to capture and test functional requirements include:

Use Case Maps (UCM): a visual notation used to describe, at a high level, how a complex system's organizational structure and emergent behavior are interconnected [17].

Unified Modelling Language (UML): a general-purpose graphical modeling language that covers structural and behavioral aspects of a system. The use case diagram (UC) is used to depict the high-level requirements of a system [18].

Test Description Language (TDL): a constructed language used to describe and therefore specify requirements as tests [19].

Testing and Test Control Notation V. 3 (TTCN-3): a standard language for test specification that is widespread and well-established [20]. A TTCN-3 module may contain a single case or several test cases that can be executed. A TTCN-3 test case can be executed against a system under test (SUT) to express its behavior. The execution's outcome is a verdict that determines if the SUT has passed the test.

4. THE MODEL-DRIVEN APPROACH

The following defines a model-driven testing approach that generates tests based on the requirements of the SUT. In many organizations, workflows are established in an ad-hoc style, with models being implicit and TCs being manually constructed. In conventional practices, software requirements are articulated in NL and then transformed into High-Level Requirement (HLR) and Low-Level Requirement (LLR) artifacts. These requirements, coupled with the implicit knowledge of the test engineer (implicit models), form the basis for the manual development of executable TCs in a proprietary testing language. The left side of Figure 1 portrays the manual workflow, while the right side represents the model-driven workflow that our proposed methodology employs.

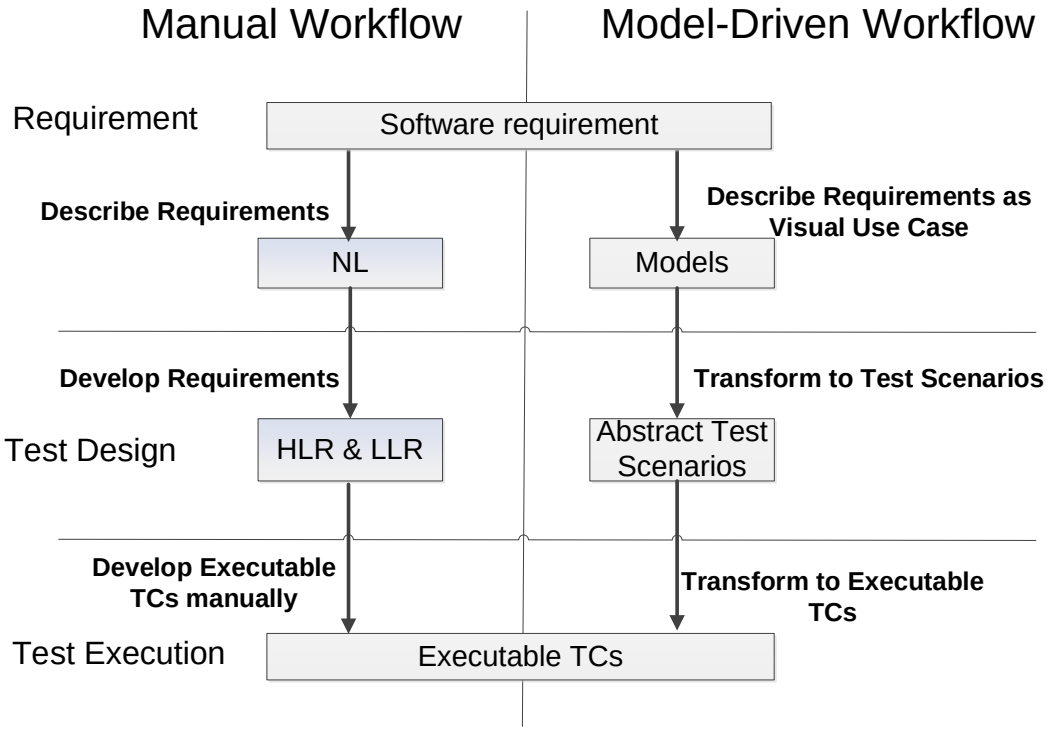


Figure 1 Manual vs Model-driven development

The new testing approach axes on three main aspects: (1) functional requirements are illustrated in visual scenario models; (2) these scenario models are subsequently transformed into test scenario descriptions; and (3) these test descriptions, further refined with test data, are ultimately converted into test cases in TTCN-3.

This approach can be viewed as a process of progressive specification refinements that involves model transformation and the integration of additional information. The primary incentive for encapsulating requirements into models is to facilitate the derivation of executable TCs.

In the subsequent subsections, we will elaborate on how model transformation and the integration of additional information are executed throughout the process to demonstrate the feasibility of this approach.

4.1. Formalizing Requirements as Scenario Models

In order to ease the transformation of NL requirements into UCM elements, the requirements are documented in Cockburn use case notation [21] and manually mapped to UCM scenario models using the jUCMNav tool [22]. Given a use case like "Withdraw Transaction", which encapsulates system behavior, the UCM scenario models can be constructed by mapping the use case elements to their corresponding UCM components and responsibility elements (this will be discussed in the next subsection).
Following is an illustration of how the behavior of "Withdraw Transaction" is formalized into a use case notation.

Primary Actor: Customer
Secondary Actor: Database (DB)
Precondition: The Customer has successfully logged into the ATM.
Postcondition: The Customer has successfully withdrawn money and received a receipt.
Trigger: The Customer opts to Withdraw Transaction.

Main Scenario:

1. DB presents the types of accounts.
2. Customer selects the type of account.
3. DB requests the withdrawal amount.
4. Customer inputs the amount.
5. Customer collects the withdrawn money.
6. DB generates and dispenses a receipt.
7. Customer collects the receipt.
8. DB shows a closing message and ejects the Customer’s ATM card.
9. Customer collects the card.
10. DB shows a welcome message.

Extensions: (Failure mode)

5a. DB informs the Customer of insufficient funds.
5b. DB provides the current account balance.
5c. DB exits the option.

4.2. Translating Use Cases into UCM Scenario Models

UCM scenario models can be constructed by mapping the Actors and Actions elements defined in the "Withdraw Transaction" use case. The mapping process is quite direct. For instance, the Primary Actor (Customer) and the Secondary Actor (DB) are manually mapped to two UCM components: Customer and DB. The actions to be carried out by each component, such as Display_Account and Choose_Account, are assigned to UCM responsibility elements. As a

general rule, the Actor elements are mapped to UCM components and the Action elements to UCM responsibility elements.

With a basic understanding of the jUCMNav tool, the Actors' actions and Actions in the use case are modeled into UCM scenarios. Figure 2 displays a UCM map comprising two components with defined responsibilities.

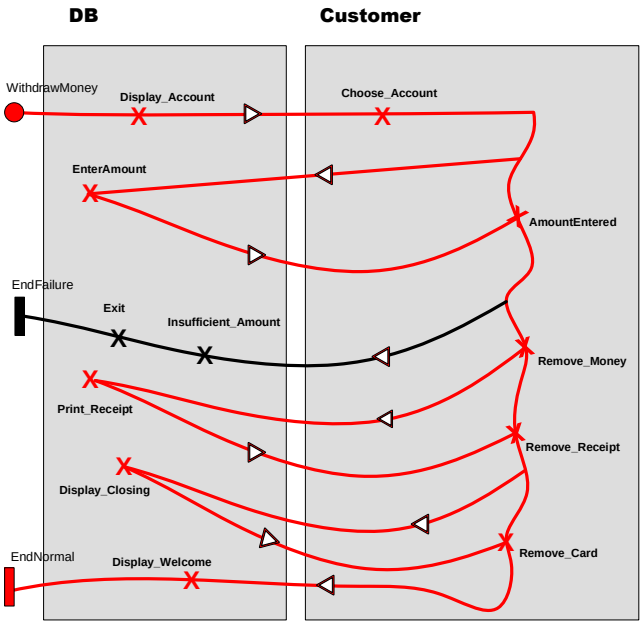


Figure 2 UCM Scenario models built from the “Withdraw Transaction” use case

4.3. Transforming Behavioral Models into Test Descriptions

Upon validating the UCM scenario model, we utilized the traversal mechanism bundled with the jUCMNav tool to flatten the scenario model into multiple scenario definitions, each of which maps scenario elements to TDL elements. The flattened scenario includes traversed UCM elements such as *Component Instance*, *Gate Instance*, *Action Reference*, *Interaction*, and so forth. The output of this traversal is several independent instances of the TDL metamodel serialized in the XMI interchange format, but without support for alternative behavior or generating concrete TDL syntax or semantics. As a result, we developed a process to transform the flattened UCM scenario model and data model (additional information) into an abstract test specification expressed as a valid TDL test specification.

The transformation process, depicted in Figure 3, employs a developed tool to parse the flattened scenario, which provides complete coverage of the UCM model, and automatically transforms it into TDL *Test Configuration* and *Test Description* elements that can be compiled into a TDL concrete syntax.

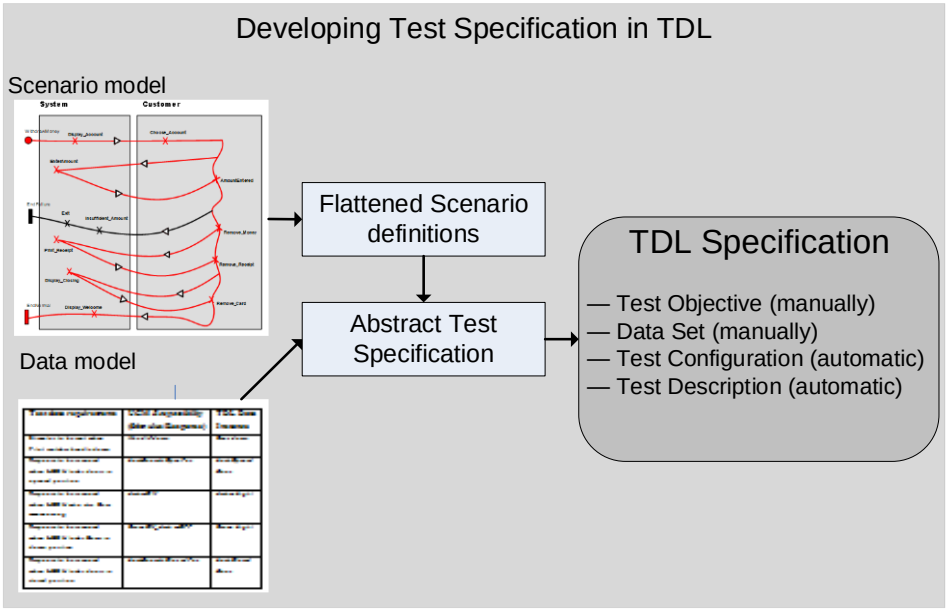


Figure 3 The TDL Test Specification Process

The tool processes the flattened scenario using an XMLStreamReader interface and automatically generates two TDL elements: *Test Description* and *Test Configuration*. The XMLStreamReader interface is utilized to iterate over the various events in the flattened scenario in order to extract information and translate it into TDL syntax. Once we finish with the current event, we proceed to the next one and continue until we reach the end of the scenario. The process transforms the flattened UCM scenario into four TDL elements. The development of each element is detailed in the following sections:

TDL Data Set: In UCM, a responsibility definition signifies an action to be performed. Leveraging this information, the responsibilities involved in a stimulus/response action can be identified as interaction messages and mapped into *Data Instances* in TDL. Procedure 1 displays compiled TDL Data Instances, grouped into two *Data Sets*, which are developed from test data.

4.3.1. TDL Data Set

A responsibility definition in UCM signifies an action that needs to be executed. Utilizing this information, the responsibilities associated with a stimulus/response action can be identified as interaction messages and mapped onto *Data Instances* in TDL. Procedure 1 displays compiled TDL *Data Instances*, which are organized into two *Data Sets*, derived from test data.

```
1.  Data Set RequestInput {
2.     instance Prompt;
3.     instance DisplayInfo;  }
4.  Data Set Preference{
5.     instance AccountType;
6.     instance Amount;
7.     instance Signal;   }
```

Procedure 1 TDL Data Sets

4.3.2. TDL Test Objective

Through the analysis of the scenario model and the integration of additional information from the system requirements, multiple *Test Objectives* can be formulated and enriched, as depicted in Procedure 2. These objectives serve as a guide for designing the *Test Description* or for designing a specific behavior.

```
1. Test Objective TestObj1 {
2. description: "Ensure that Customer selects an
   account type within 15 seconds";}
3. Test Objective TestObj2 {
4. description: "Ensure that Customer removes the
   card within 15 seconds"; }
```

Procedure 2 TDL Test Objective

4.3.3. TDL Test Configuration

The information exchange between the Tester and the SUT components is conducted through a communication point (Gate). Therefore, the *Test Configuration* in TDL encompasses all the elements required for this communication, such as *Component Instances* which could be a part of either a Tester or a SUT, and Connections. Procedure 3 demonstrates the TDL *Test Configuration*, which is automatically generated from the exported "WithdrawTransaction" scenario.

```
1.  Gate Type defaultGT accepts RequestInput, Preference;
2.  Component Type defaultComp { gate types :defaultGT ; }
3.  Test Configuration TestConfiguration {
4.    //Customer component
5.    instantiate Customer as Tester of type
      defaultComp having { gate gCustomer of type defaultGT ; }
6.    //DB component
7.    instantiate DB as SUT of type defaultComp
8.    having { gate gDB of type defaultGT ;  }
9.    //connect the two components through their gates
10.   connect gCustomer to gDB;  }
```

Procedure 1 TDL Test Configurated generated from the “Withdraw Transaction” Scenario

4.3.4. TDL Test Description

The *Test Description* in TDL outlines the anticipated behavior, actions, and interactions between DB components. The TDL *Action* element to be executed corresponds to its equivalent, the UCM responsibility object. On the other hand, the TDL *Interaction* element represents a message that is sent from a source and received by a target. Procedure 4 illustrates the TDL *Test Description*, which is composed of actions, timers, and interactions.

```
1.  Test Description TestDescription {  //Test description definition
2.  use configuration: TestConfiguration;  {
3.  perform action Display_Account on component DB ;
4.  perform action Choose_Account on component Customer  };
5.  gDB sends instance Prompt to gCustomer with { test objectives: TestObj1;};
6.  gCustomer sends instance AccountType to gDB with { test objectives: TestObj1;   };
7.  gDB sends instance Prompt to gCustomer  };
8.  gCustomer sends instance Amount to gDB };
9.  perform action CheckAmount on component DB ;
10. gDB sends instance DisplayInfo to gCustomer  with { test objectives :TestObj2;  }; };
11. perform action Remove_Money on component Customer ;
12. perform action Remove_Card on component Customer ;
13. repeat 2 times {  //Iterate over receiving responses,
14. alternatively  {
15.   gDB sends instance Prompt to gCustomer with  { test objectives: TestObj1; };
16.     set verdict to PASS ; }
17.     or { gate gCustomer is quiet for (15.0 SECOND);
18.     set verdict to FAIL;  }
19.  alternatively  {  // Customer sends Amount
20.   gCustomer sends instance Amount to gDB;
21.    set verdict to PASS ;    }
22.    or { Amount < Balance; }
23.    set verdict to FAIL;  }  }  }
```

Procedure 2 TDL Test Description

As previously stated, the lack of an alternative element in the scenario metamodel necessitated manual modification of the generated *Test Description* to merge the scenarios that represent alternate test behavior. Ultimately, the four TDL elements are consolidated into a single TDL Test Specification and used as a foundation to generate an executable test in a scripting language.

4.4. Transforming TDL Specification into Test Cases in TTCN-3

The conversion of the TDL Specification model into an executable test in TTCN-3 is carried out based on their metamodels and transformation rules that we defined in the form of mappings between the elements of the metamodels. The transformation rules, as displayed in Table 1, are programmed and implemented using a model-to-text technology tool called Xtend.

Table 1 Displays the Transformation Rules between TDL and TTCN-3.

Rule #	TDL Meta-model elements (syntax)	Our TDL concrete syntax	Equivalent TTCN-3 statements	Descriptiom
1	TestConfiguration	Test Configuration <tc_name>	module <tc_name> { }	Map to a module statement with the name < td_name >
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.
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.
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.
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
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.
7	AlternativeBehaviour	alternatively { }	alt {}	Map to an alt statement
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>	<comp_name1>	Map to a receive statement

		sends instance <instance_Inx> to <comp_name2>	.receive(<instance_In X>)	that receives a response when the sending source is a SUT component.
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
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
11	VerdictAssignme nt	set verdict to <verdict_value>	setverdict (<verdict_value>)	Map to a setverdict statement.
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.
13	Stop	stop	stop	Map to a stop statement within an alt statement.
14	Break	break	break	Map to a break statement within an alt statement.
15	Timer	timer <timer_name>	timer<timer_name>	Map to a timer definition statement.
16	TimerStart	start <timer_name> for (time_unit)	<timer_name>.start(ti me_unit);	Map to a start statement.
17	TimerStop	stop <timer_name>	<timer_name>.stop;	Map to a stop statemen.t
18	TimeOut	<timer_name> times out	<timer_name>.timeou t;	Map to a timeout statement.
19	Quiescence/ Wait	is quite for (time_unit) waits for (time_unit)	timer <timer_name> <timer_name>.start(ti me_unit); <timer_name>.timeou t	Map to a timer definition statement, a start statement and to a timeout statement.
20	InterruptBehaviou r	interrupt	stop	Map to stop statement
21	BoundedLoopBeh aviour	repeat <number> times	repeat	Map to a repeat statement. The repeat is used as the last statement in the alt behaviour. It should be used once for each possible alternative.
22	DataSet	Data Set <DataSet_name> { }	type record <DataSet_nameType > { }	Map Data Set to record type using DataSet_name and prefixed with “Type”
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

The TTCN-3 Test Description module transformed from the TDL Specification is shown in

Procedure **3**.

```
1.  module TestDescription {
2.    import from TestConfiguration all;
3.    import from WithdrawData all;
4.    testcase _TC () runs on Customer {
5.    map (mtc:gCustomer, system:gDB);
6.    timer SelectionTime; timer RemoveTime;
7.    Display_Account ();  // function call
8.    gDB.send(PromptTemplate);
9.    SelectionTime.start(15.0);
10.   alt {
11.     [ ] gDB.receive(ChoiceTemplate) {
12.        SelectionTime.stop;
13.       Set verdict(pass);
14.       CheckAmount ();  // function call
15.       gDB.send(DisplyInfoTemplate);
16.      RemoveTime.start(15.0);
17.      repeat } // restart the alt
18.       RemoveTime.timeout {
19.       setverdict(fail) }
20.     [ ] gDB.receive(SignalTemplate) {
21.       RemoveTime.stop;
22.       setverdict(pass); }
23.    }
24.   unmap (mtc:gCustomer, system:gDB);  }  }
25.   function Display_Account () runs on DB {  }
26.   function CheckAmount () runs on DB {  }
27. }
```

Procedure 3 Test case in TTCN-3

5. ASSESSMENT AND OUTCOME OF THE APPROACH

We assessed the approach using a private case study that served as our SUT. We employed three legacy use cases expressed in NL that describe the behavior of an avionics product. The experiment began by encapsulating the requirements into UCM scenario models. Once the

scenario models were validated, they were used as input and transformed into *test descriptions*, which were subsequently converted into executable test cases in TTCN-3. The details of the experiment are presented in the following paragraph.

The execution of the three use cases using the new testing process resulted in the generation of 26 test scenarios and test cases. The new testing process covered all paths in the scenario models, generating one test case for each scenario path, as outlined in Table 2.

Table 2 Coverage Rate Achieved by the New Testing Process

use case modelled 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

5.1. Proficiency Regarding the Fulfillment of Specifications and Production of Accurate Test Scenarios

We conducted an evaluation of the produced TCs to ensure they adequately encompass the requirements. As demonstrated in Table 2, the method effectively covered all routes within the scenario models. Indeed, the method yielded one TC per every pathway in the scenario model. The total count of the produced TCs successfully encompasses all potential routes in the UCM model, thereby achieving comprehensive scenario and requirement coverage.

The correctness of the generated test cases was evaluated by comparing their test outcomes with those of the legacy tests. The goal is to align the behavior of the test cases with that of the legacy tests. As stated, the legacy tests serve as a benchmark for evaluating the accuracy of the generated test cases. Table 3 exhibits the results of the verdict comparison for each test case pair. The scenario models that articulate the requirements are displayed in the first column, followed by a description of the test case in the second column. The third column presents the rate of verdict alignment with the corresponding legacy test.

Table 3. The Correspondence Percentage of the Implemented Test Cases.

use case modelled 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	ETAComputation	98 %

All the outcomes in the *Fly-by-procedure* and *Fly-over-procedure* test cases corresponded with the respective outcomes of the legacy tests. In the remaining test cases, which include *Fly-over-procedure via DES+SAR*, *Manual Direct to-Intercept* and *ETA computation*, only a minimal number of outcomes did not align with the corresponding legacy tests. The results in the third column confirmed a high success rate in determining the SUT behavior—producing a pass verdict when expected and a fail verdict when errors are present.

5.2. Discussion

The validation was achieved by comparing the behavior of the legacy and the generated tests. If they exhibit equivalent behavior, meaning they have the same sequence of test events and verdicts, they can be considered comparable. Almost all verdicts of oracle steps in the generated test cases matched their corresponding ones in the legacy tests. Essentially, the generated test cases passed and failed at the same steps as the legacy test cases, with a few exceptions of failures in the generated tests, predominantly due to timing issues.

The tests generated for TTCN-3 execution demonstrated significantly better performance compared to the legacy system, as the SUT is relatively slow. These cases could be easily identified by examining the state of the SUT. If the state remained the same as the preceding test event, it indicated that the SUT had not yet updated its state. In this context, the responses are not immediate; instead, the test system must query the SUT to receive the response. Moreover, some of the failures could suggest the existence of alternative behavior in the SUT, something the legacy test system couldn't manage due to its reliance on linear sequences of test events.

In conclusion, this study demonstrates that our approach generated test cases that fully covered all the requirements described in the scenario models. When compared to the legacy testing system, the new approach enhances practical testing and offers several benefits to test engineers.

The advantages of our new testing practice include:

— **Enhanced understanding of the test system**: Utilizing a model provides an overview of the system's behavior as opposed to scattered pieces of information.
— **Early Testing**: Test engineers don't need to wait; they describe the requirements in a model and generate the tests at the press of a button.
— **Reduced testing effort**: In our model-driven testing, the number of iterations needed to produce correct test cases is reduced. The test development phase is eliminated. Test cases are no longer manually written or corrected, but generated.
— **Traceability**: Documentation can be produced from the model, ensuring consistency with the tests. Since test cases are derived from the UCM models where requirements are described, any defect found during test case execution can be traced back to its requirement.
— **Systematic and automated**: With the aid of the developed tools, repeated tests are possible, ensuring robustness of the test results.
— **Reduced human errors**: As tests are generated from the model and thus consistent with requirements, the likelihood of errors in the test suite is inherently reduced.

6. GENERALIZATION OF THE APPROACH

Our approach primarily targets the functional aspects of software and has been applied to two realistic case studies in the avionics domain. Furthermore, the methodology is potentially applicable to safety-critical software, as it addresses timing requirements and provides

traceability from requirements to tests. The approach leverages two key elements to enhance the testing process:

Modeling: The system's functional requirements and design are represented through high-level visual models and DSL, abstracting away from technological implementation details.
Model transformation: Automated model transformations are employed to generate tests, reducing manual labor and enabling the simulation of high-level models to validate the appropriateness of the modeled system behavior at an early stage of development.

In the present day, the practical implementation of model-driven testing benefits from a range of tools and technologies. Certain requirements, such as robustness requirements, may not be expressible with UCM notation and would need to be specified using other notations or languages. The model transformations are largely automated and require minimal human intervention. The process transforms informal requirements into a formal UCM model. We utilized the tool described in [23] that generates individual test traces, referred to as test scenarios in TDL. However, it's important to note that test traces aren't always test cases. A quality test case includes alternative behavior in both TDL and TTCN-3. This portion had to be manually generated as no tools currently exist to perform this task. The tips found in [23] were tested and proved successful, indicating that the implementation of this part of the translator is a future task. Nevertheless, the translation from TDL to TTCN-3 is fairly straightforward, as there is mostly a one-to-one mapping from TDL to TTCN-3. Only aspects like describing test purposes aren't covered and therefore have to be manually translated, typically as TTCN-3 comments.

In summary, our accomplishment was to demonstrate the benefits of constructing a formal UCM model, as everything downstream can be automatically generated and is either entirely correct or entirely incorrect. Test automation has the advantage of being systematic in handling errors, in contrast to manual processes where errors are introduced randomly and are challenging to trace. This automation reduces the amount of manual work required for test development, making the testing process less prone to errors and more efficient.

7. LESSONS LEARNED

We've gathered some significant insights from the development and implementation of the testing methodology. Users of the methodology shouldn't need to have functional requirements expressed in use case notation to model them as scenarios. However, having requirements presented as use cases did facilitate the mapping to UCM models.

The model transformation to the TDL domain isn't completely automatic and requires human intervention to obtain data elements and construct alternatives. The TDL models were a crucial part of model-driven testing as they were used as both inputs and outputs in the model transformation process. The decision to use TDL notation in the development of tests proved to be successful. TDL effectively bridged the gap between the described requirements and tests, serving as a means of communication with non-technical individuals and as a foundation for generating concrete tests.

8. CONCLUSIONS

Manually translating software requirements, expressed in natural language, into executable tests while ensuring adequate test coverage can be a laborious and error-prone process. Improvements in the testing process can be achieved by generating tests based on behavioral models and model

transformation. Model-driven testing leverages transformation techniques to produce test artifacts by interrelating models at different abstraction levels.

This study proposes a model-driven testing strategy to create executable test cases from visual use cases. The approach begins by outlining the SUT requirements in use case models, which are then transformed and mapped to abstract test scenarios. These scenarios are further refined and transformed into executable tests in a scripting language. The approach was implemented and evaluated in an industrial case study. Our ongoing work in this field aims to achieve complete automation of the proposed method where possible, with a special focus on automating the merging of linear scenarios (identifying common paths up to a UCM branch).

REFERENCES

[1] 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).

[2] 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).

[3] Kesserwan, Nader, et al. "From use case maps to executable test procedures: a scenario-based approach." Software & Systems Modeling 18.2 (2019): 1543-1570.

[4] Kesserwan, N. (2020). Automated Testing: Requirements Propagation via Model Transformation in Embedded Software (Doctoral dissertation, Concordia University).

[5] Nogueira, S., Sampaio, A., & Mota, A. (2014). Test generation from state-based use case models. Formal Aspects of Computing, 26(3), 441-490.

[6] Sarmiento, E., Sampaio do Prado Leite, J. C., & Almentero, E. (2014, August). C&L: Generating model-based test cases from natural language requirements descriptions. In Requirements Engineering and Testing (RET), 2014 IEEE 1st International Workshop on (pp. 32-38). IEEE

[7] Somé, S. S., & Cheng, X. (2008, March). 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.

[8] Heckel, R., & Lohmann, M. (2003). Towards model-driven testing. Electronic Notes in Theoretical Computer Science, 82(6), 33-43. ISBN 1571-0661.

[9] J. Ryser and M. Glinz "A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts" Proc. 12th Int'l Conf. Software and Systems Eng. and Their Applications, Dec. 1999.

[10] Kesserwan, N., Dssouli, R., & Bentahar, J. (2018). Modernization of Legacy Software Tests to Model-Driven Testing. In Emerging Technologies for Developing Countries: First International EAI Conference, AFRICATEK 2017, Marrakech, Morocco, March 27-28, 2017 Proceedings 1st (pp. 140-156). Springer International Publishing.

[11] J. Wittevrongel, Maurer, F., SCENTOR: Scenario-Based Testing of EBusiness Applications, Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises, 2001, pp. 41 - 46.

[12] F. Fraikin, Leonhardt, T., SeDiTeC — Testing Based on Sequence Diagrams, 17th IEEE International Conference on Automated Software Engineering, 2002, pp. 261 - 266.

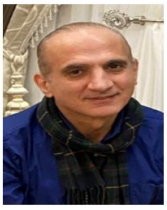
[13] Gagarina, Larisa G., Anton V. Garashchenko, Alexey P. Shiryaev, Alexey R. Fedorov, and Ekaterina G. Dorogova. "An approach to automatic test generation for verification of microprocessor cores." In 2018 IEEE Conference of Russian Young Researchers in Electrical and Electronic Engineering (EIConRus), pp. 1490-1491. IEEE, 2018.

[14] L. C. Briand, Labiche, Y., A UML-Based Approach to System Testing, 4th International Conference on the Unified Modeling Language (UML), Toronto, Canada, 2001, pp. 194-208.

[15] F. Basanieri, Bertolino, A., Marchetti, E., The Cow_Suite Approach to Planning and Deriving Test Suites in UML Projects, Proceedings of the 5th International Conference on The Unified Modeling Language, Springer-Verlag, 2002, pp. 383-397.

[16] W. Grieskamp, Nachmanson, L., Tillmann, N., Veanes, M., Test Case Generation from AsmL Specifications - Tool Overview, 10th International Workshop on Abstract State Machines, Taormina, Italy, 2003
[17] Buhr, Raymond JA. "Use case maps as architectural entities for complex systems."Software Engineering, IEEE Transactions on 24.12 (1998): 1131-1155.
[18] Fowler, Martin. UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional, 2004.
[19] ETSI ES 203 119 (stable draft): Methods for Testing and Specification (MTS); The Test Description Language (TDL).
[20] http://www.ttcn-3.org/index.php/downloads/standards.
[21] Adolph, S., Cockburn, A., & Bramble, P. (2002). Patterns for effective use cases. Addison-Wesley Longman Publishing Co., Inc.
[22] http://istar.rwth-aachen.de/tiki-index.php?page=jUCMNav.
[23] Boulet, P., Amyot, D., & Stepien, B. (2015). 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 International Publishing.

AUTHORS



Nader Kesserwan joined Robert Morris University in 2020 as an assistant professor of software engineering after several years of academic and industrial experience. Dr. Kesserwan finished his Ph.D. in Information Systems and Engineering at Concordia University, Montreal, Canada. His industrial experience includes R&D activities in Avionics; testing embedded systems such as flight simulators and flight management systems. His research interests centre around requirements engineering and model-driven testing. Dr. Kesserwan published several journal papers in software engineering and participated in several conferences.

Jameela Al-Jaroodi received her Ph.D. in computer science from the University of Nebraska–Lincoln, Nebraska, USA and an M.Ed. in higher education management from the University of Pittsburgh, Pennsylvania, USA. She was a Research Assistant Professor at Stevens Institute of Technology, Hoboken, NJ, USA, then an Assistant Professor at the United Arab Emirates University, UAE. Then she was an independent Researcher in the computer and information technology. She is currently an Associate Professor and Coordinator of the software engineering concentration at the Department of Engineering, Robert Morris University, Pittsburgh, Pennsylvania, USA. She is involved in various research areas, including middleware, software engineering, security, cyber-physical systems, smart systems, and distributed and cloud computing, in addition to UAVs and wireless sensor networks.

Nader Mohamed is an associate professor in the Department of Computing and Engineering Technology, Pennsylvania Western University, California, Pennsylvania, USA. He teaches courses in cybersecurity, computer science, and information systems. He was a faculty member with Stevens Institute of Technology, Hoboken, NJ, USA and UAE University, Al Ain, UAE. He received the Ph.D. in computer science from the University of Nebraska–Lincoln, Lincoln, NE, USA. He also has several years of industrial experience in information technology. His current research interests include cybersecurity, middleware, Industry 4.0, cloud and fog computing, networking, and cyber-physical systems.

Imad Jawhar is a professor at the Faculty of Engineering, Al Maaref University, Beirut Lebanon. He has a BS and an MS in electrical engineering from the University of North Carolina at Charlotte, USA, an MS in computer science, and a Ph.D. in computer engineering from Florida Atlantic University, USA, where he also served as a faculty member for several years. He has published numerous papers in international journals, conference proceedings and book chapters. He worked at Motorola as engineering task leader involved in the design and development of IBM PC based software used to program the world's leading portable radios, and cutting-edge

communication products and systems, providing maximum flexibility and customization. He was also the president and owner of Atlantic Computer Training and Consulting, which is a company based on South Florida (USA) that trained thousands of people and conducted numerous classes in the latest computer system applications. Its customers included small and large corporations such as GE, Federal Express and International Paper. His current research focuses on the areas of wireless networks and mobile computing, sensor networks, routing protocols, distributed and multimedia systems. He served on numerous international conference committees and reviewed publications for many international journals, conferences, and other research organizations such as the American National Science Foundation (NSF). He is a member of IEEE, ACM, and ACS organizations.