PROPAGATION OF SOFTWARE REQUIREMENTS TO EXECUTABLE TESTS

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 start at the beginning of testing as abstract requirements representing the system behavior. The manual development of these test cases is labor-intensive, error-prone, and costly. Describing the system requirements into behavioral models and transforming them into a scripting language has the potential to automate their propagation to executable tests. Ideally, an efficient testing process should begin as early as possible, refine the use cases with sufficient details and facilitate test case creation. We propose an approach that enables automation in propagating functional requirements to executable test cases through model transformation. The proposed testing process begins with capturing system behavior as visual use cases, adopting a domain-specific language, defining transformation rules, and finally transforming the use cases into executable tests.

KEYWORDS

Model-Driven, Model Transformation, Requirement Propagation, & Test Cases

1. INTRODUCTION

Nowadays, software development has become increasingly complex, resulting in high demand for software verification. Adopting an unsuitable test methodology would compromise system safety. For instance, the avionics sector has observed exponential growth in safety-critical software, military or civil. Some of the challenges that test engineers face are time (lack of time for detailed calculations). During software development, the manual creation of test artifacts remains a major cost factor consuming more than 50% of the overall development effort [1]. Enable automation in the testing process can increase software quality and ensure the robustness of the test results leading to a reduction in liability cost and human effort. In software engineering, scenarios have become popular to elicit, document, and validate requirements [2]. Scenarios illustrate a sequence of actions related to behavior and help reduce the complexity of an application for a better understanding and prioritizing of the desired scenario. The system requirements, functional and operational requirements, are captured in the form of scenarios and used to determine test cases (TCs).

A scenario provides details about how to implement behavior reasonably. Since scenarios of a system together represent its behavior and its functionality domain, they maintain statement and decision coverage when a system is divided into test scenarios. When requirements are modeled

David C. Wyld et al. (Eds): SOEN, SIPP, PDTCA, ITE, CCSIT, NLPCL, DaKM, BIGML, AISC -2023
pp. 33-46, 2023. CS & IT - CSCP 2023 DOI: 10.5121/csit.2023.131203

with use cases, a scenario can take a specific path through the model to instantiate a use case [3], [4]. As a result, selecting a use case for testing purposes, all its possible test scenarios can be driven to satisfy a branch coverage criterion. The execution of these test scenarios on the system under test (SUT) can measure the completeness of the path coverage criterion. Consequently, these path coverage metrics address the safety requirements and help test engineers find redundant or missing test scenarios.

Our goal is to explore an alternative testing methodology that supports test automation and starts with representing requirements, formalizing test descriptions independently of the test scripting language, and targeting a testing language. Enabling automation in the testing process has the potential to reduce test effort, reduce human errors, start testing early, and support the auto-generation of the testing artifacts.

We were motivated to build a testing methodology that can use scenario-based notations to support the derivation of the TCs and make use of standard notations to capture and test functional requirements.

This paper is organized as follows: Section 2 surveys related work; Section 3 provides background information. Section 4 presents the approach followed by an evaluation of this approach in Section 5, and Section 6 concludes and draws future work directions.

2. RELATED WORK

Several methods were proposed in the literature to generate test cases from use cases expressed in NL and UML diagrams. The following papers highlight a few that are relevant to this research.

Ryser and al. [5] propose the SCENT method to create scenarios based on NL requirements. These scenarios are formalized in state charts and enriched with additional information. The state charts are flattened by a traversal algorithm to determine the test cases. More tests are modeled in dependency charts and generated from the dependencies between scenarios increasing the testing effort.

Sarmiento et al. [6] developed a tool that models NL requirements using UML activity diagrams. Their approach generates test cases based on the activity diagrams to support the automated testing. The drawback of the technique is the requirement of using lexicon symbols to reference relevant words.

Heckel et al. [7] proposed an MDT approach that separates the generation of test cases from their execution on different target platforms. This approach for testing applications is designed in a model-driven development context. However, the application of the approach is limited to generating test cases in a model-driven context.

Somé et al.[8] proposed an approach that specifies use cases with restricted NL to be mapped later to finite state machine models (FSM). A traversal algorithm navigates through the FSM models and generates test scenarios based on a coverage criterion. However, the approach needs to modify the use cases to the restricted NL and create FSM diagrams which are considered an overhead.

Nogueira et al. [9] proposed a method that captures requirements in a standard way using document templates. This automatic test generation approach extends the templates to allow inclusion and extension relations between use cases. Furthermore, the technique permits the

inclusion of data elements as parameters, user-defined types, and variables. However, the approach doesn't generate executable test cases as the templates that capture control flow, state, input, and output as a source are used to generate formal models only.

Some approaches as in [10], [11], and [12] use implicit relationships to support test generation, execution, and evaluation; while others like in [13] use implicit relationships to support regression testing. Further approaches use explicit relationships to support test generation [14], test execution and evaluation [15], or coverage analysis.

Like most methods mentioned above, our work relates to them and derives test cases from use cases. However, it differs by formalizing the requirements as abstract test scenarios and transforming them into executable test cases. This separation of test specification from test implementation results in more flexibility for test deployment and allows test engineers to focus on the objectives of the test.

The approach provides scenario coverage criteria and allows prioritizing desired requirements to be tested early in the process

3. BACKGROUND

In software development, several approaches have been developed to modernize testing activities. A known technique is model-driven testing which is an automated way of transforming and creating models based on transformation rules defined in terms of mappings between the elements of metamodels. In this model transformation engineering, the models at various levels of abstraction are used to enable generalization and automated development.

Another technique used to modernize the testing processes is specification-based testing (SBT). The approach has been applied in the testing of complex software systems. SBT demonstrates that the software or the system meets the requirements and provides value. Thus, test engineers enjoy the precision and the detailed description of the system requirement offered by the SBT approach. The tests are developed from the context of the requirements and designed from the user's perspective, not the designer's.

There are several modeling languages to express functional requirements and numerous languages that can be used to specify or describe TCs. Those TCs can be developed manually or derived automatically from behavior models. Some of the modeling notations used to capture and test functional requirements are:

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 [16].

Unified Modelling Language (UML): is a graphical general-purpose modeling language that covers structural and behavioral aspects of a system. The use case diagram (UC) is used to describe the high-level requirements of a system [17].

Test Description Language (TDL): a constructed language to describe, and thus specify requirements as tests [18].

Testing and Test Control Notation V. 3 (TTCN-3): a standard language for test specification that is widespread and well-established [19]. 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 SUT to express its behavior. The result of execution is a verdict that determines if the SUT has passed the test.

4. THE REQUIREMENT PROPAGATION APPROACH

Define a model-driven testing approach that generates tests based on the requirements of the SUT. Several organizations develop their workflow in an ad-hoc fashion where models are implicit, and Test cases (TCs) are written manually. The software requirements in the legacy practice are expressed in Natural Language (NL) and are developed into High-Level Requirement (HLR) and Low-Level Requirement (LLR) artifacts. These requirements are subsequently used as the basis, along with test engineer knowledge (implicit models), for developing manually executable TCs written in a proprietary test language. The left side of Figure 1 shows the manual workflow, while the right side shows the model-driven workflow adopted by our proposed methodology.

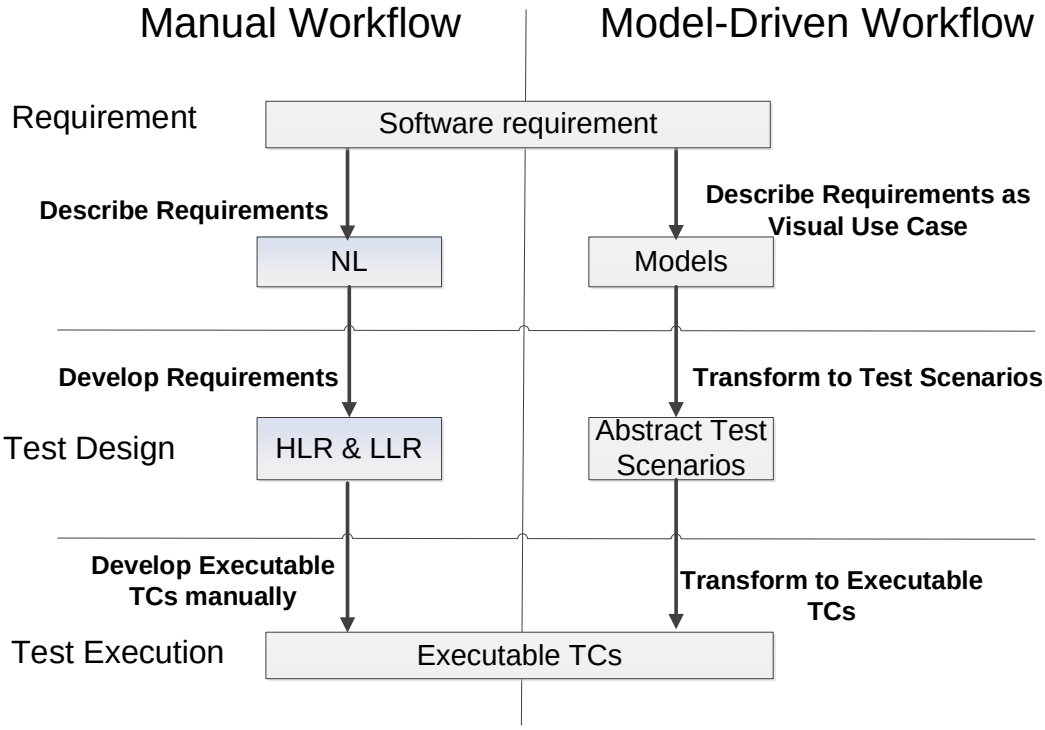


Figure 1 Manual vs Model-driven development

The main points of the new testing approach are: (1) functional requirements are described in visual scenario models; (2) the scenario models, in their turn, are transformed to test scenario descriptions; and (3) the resulting test descriptions, refined with test data, are finally transformed to test cases 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 basic motivation for capturing requirements into models is to support the derivation of executable TCs.

In the following subsections, we explain how model transformation and insertion of additional information are performed throughout the process to demonstrate the approach's feasibility.

4.1. Formalizing Requirements as Scenario Models

To facilitate the modeling of the NL requirements into UCM elements, the requirements are written in Cockburn use case notation [20] and mapped manually to UCM scenario models using the jUCMNav tool [21]. Given a use case: "Withdraw Money" that captures system behavior, the UCM scenario models can be built by mapping the use case elements to their equivalent UCM components and responsibility elements (discussed in the next subsection).

The following shows how the behavior of the "Withdraw Money" is formalized into a use case notation.

Primary Actor: Bank Customer
Secondary Actor: DB.
Precondition: The Customer has logged into the ATM.
Postcondition: The Customer has withdrawn money and received a receipt.
Trigger: The Customer has chosen to withdraw money.
Main Scenario:
1. DB displays account types.
2. Customer chooses account type.
3. DB asks for the amount to withdraw.
4. Customer enters the amount.
5. Customer removes money.
6. DB prints and dispenses receipts.
7. Customer removes receipt.
8. DB displays a closing message and dispenses the Customer's ATM card.
9. Customer removes the card.
10. DB displays a welcome message.

Extensions: (Failure mode)
5a . DB notifies the Customer that funds are insufficient.
5b. DB gives the current account balance.
5c. DB exits option.

4.2. Mapping the use case to UCM scenario models

UCM scenario models can be built by mapping the Actors and the Actions elements defined in the "Withdraw Money" use case. The mapping is straightforward, for example, the Primary Actor (Customer) and the Secondary Actor (DB) are mapped manually to two UCM components: Customer and DB. The actions to be performed by each component, such as Display_ Account and Choose_Account are allocated to UCM responsibility elements. As a rule, the Actors' elements are mapped to UCM components and the Actions' elements to UCM responsibility elements.

With some basic knowledge of the jUCMNav tool, the Actor's ions and Actions in the use case are modeled into UCM scenarios. Figure shows a UCM map consisting of two components with bounded responsibilities.

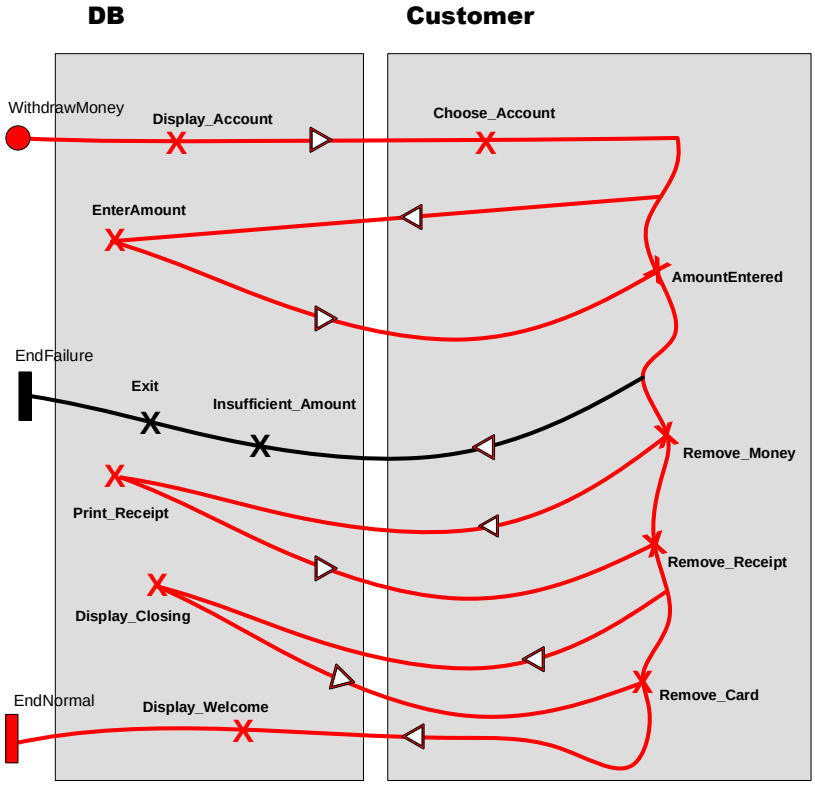


Figure 2 UCM Scenario models built from the "Withdraw Money" use case

4.3. Transform UCM Scenario Models into Test Descriptions

After validating the UCM scenario model, we used the traversal mechanism packaged with the jUCMNav tool to flatten the scenario model to several scenario definitions where each scenario element is mapped to TDL elements. The flattened scenario contains traversed UCM elements such as *Component Instance, Gate Instance, Action Reference, Interaction, etc*. The result of this traversal is several independent instances of TDL metamodel serialized in the XMI interchange format without support for alternative behavior or producing concrete TDL syntax or semantics. Therefore, 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, as shown in Figure 3, uses a developed tool to parse the flattened scenario, which has complete coverage of the UCM model, and transform it automatically to TDL *Test Configuration* and *Test Description* elements that can be compiled into a TDL concrete syntax.

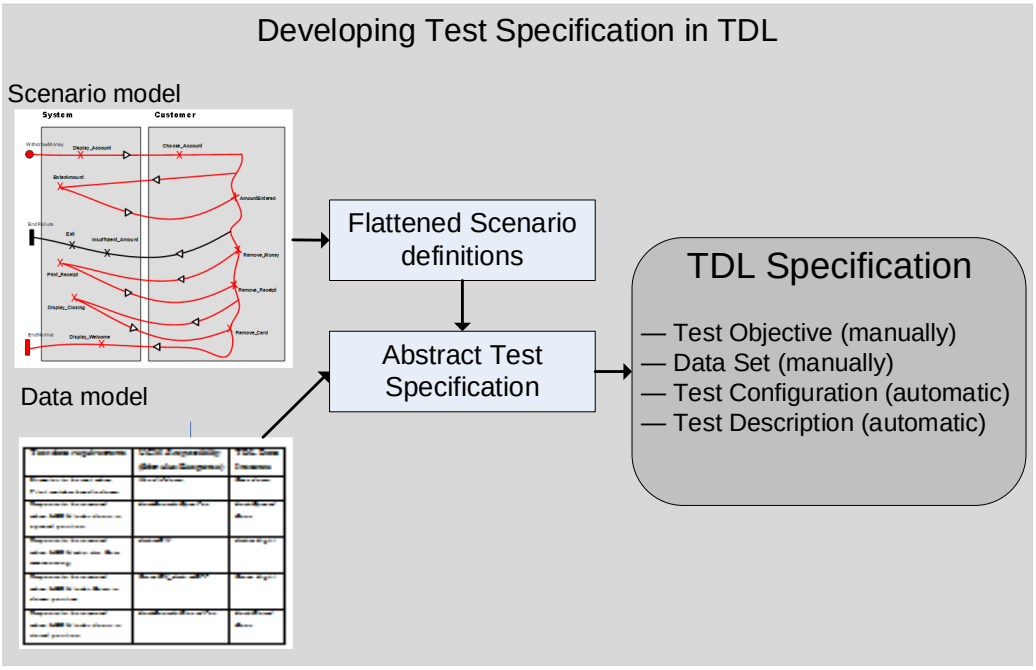


Figure 1 The TDL Test Specification Process

The tool parses the flattened scenario using an 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 flattened 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 the end of the scenario. The process transforms the flattened UCM scenario into four TDL elements. The development of each element is shown in the following:

TDL Data Set: A *responsibility* definition in UCM represents an action to perform. Using this information, the *responsibilities* involved in a stimulus/response action can be flagged as interaction messages and mapped into *Data Instances* in TDL. Algorithm 1 shows compiled TDL *Data Instances* grouped in two *Data Sets* that are developed from test data.

4.3.1. TDL Data Set

A *responsibility* definition in UCM represents an action to perform. Using this information, the *responsibilities* involved in a stimulus/response action can be flagged as interaction messages and mapped into *Data Instances* in TDL. Algorithm 1 shows compiled TDL *Data Instances* grouped in two *Data Sets* that are developed 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;   }
```

Algorithm 1 TDL Data Sets

4.3.2. TDL Test Objective

By analyzing the scenario model and adding additional information from the system requirements, several Test Objectives can be developed and enriched as shown in
Algorithm . These objectives are used as guidelines to design the Test Description or to design a particular 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"; }
```

Algorithm 2 TDL Test Objective

4.3.3. TDL Test Configuration

The exchange of information between the Tester and SUT components is performed via a point of communication (Gate). As such, the Test Configuration in TDL contains all the elements required for this communication, such as Component Instances may either be a part of a Tester or a part of a SUT and Connections. Algorithm 3 shows the TDL Test Configuration generated automatically from the exported "WithdrawMoney" 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;  }
```

Algorithm 3 TDL Test Configurated generated from the "Withdraw Money" Scenario

4.3.4. TDL Test Description

The Test Description in TDL defines the expected behavior, the actions, and the interactions between DB components. The TDL Action element to be performed is matched with its equivalence, the UCM responsibility object. Whereas, the TDL Interaction element represents a message sent from a source and received by a target. Algorithm 4 shows the TDL Test Description 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;  } } }
```

Algorithm 4 TDL Test Description

As mentioned earlier, the absence of an alternative element in the scenario metamodel required manual adjustment of the generated *Test Description* to merge the scenarios that constitute alternate test behavior. Finally, the four TDL elements are combined in one TDL Test Specification and used as a base to generate an executable test in a scripting language.

4.4. Transform TDL Specification into Executable Test

The transformation of the TDL Specification model to an executable test in TTCN-3 is done based on their metamodels and transformation rules that we defined in terms of mappings between the elements of meta-models. The transformation rules as shown in Table 1, are programmed and implemented in a tool based on model-to-text technology called Xtend.

Table 1 shows the transformation rules from TDL to TTCN-3.

Rule	TDL Meta-model elements (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 a 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
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 statemen.t
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 behaviour. 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

The TTCN-3 Test Description module transformed from the TDL Specification is shown in Algorithm 5.

```
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. }
```

Algorithm 5 Executable Testcase in TTCN-3

5. EVALUATION AND RESULT OF THE APPROACH

We evaluated the approach with a private case study that we used as a SUT. We used three legacy use cases expressed in NL that describe the behavior of an avionics product. The conducted experiment started by capturing the requirements into UCM scenario models. Once the scenario models were validated, the approach used them as input and transformed them into test descriptions which later transformed into executable test cases in TTCN-3. The details of the experiment are shown in the following paragraph.

The execution of the three use cases on the new testing process resulted in generating 41 test scenarios and TCs. The new testing process covered all paths in the scenario models generating one TC for each scenario path, as listed in Table 2.

Table 2 The coverage rate achieved by the new testing process

use case	Scenario Path		# of TCs	Requirement Coverage Rate
	Primary	Secondary		
Use case 1	7	11	18	100 %
Use case 2	3	9	12	100 %
Use case 3	4	7	11	100 %
Total	**14**	**27**	**41**	

We analyzed the generated scenario models and found that they covered the requirements and, in their turn, were transformed to correct test cases when executed against the SUT. As a result, the new testing process achieved the scenario and requirement coverage criterion and generated correct TCs. We found the following benefits from our new testing practice:

Visual representations: capture requirements in a model provides visual representations to test engineers and increase their understanding as opposed to scattered information.

Early Testing: Once software requirements are described in scenario models; the test cases can be generated by pressing the button.

Decreased test effort: in our auto-generation testing approach, the number of cycles to get correct test cases is reduced. The test development phase is generated instead of manually developed.

Reduced human errors: The fact that the tests are generated at will from the model reduces the possibility of error and ensures the robustness of the test results.

6. CONCLUSIONS

The manual propagation of software requirements, expressed in natural language, to executable tests and ensuring acceptable test coverage is labor-intensive and error-prone. Generating tests based on behavioral models and model transformation has the potential to improve the testing process. Model-driven testing uses the transformation technique to generate test artifacts by mapping models and linking them together at different levels of abstraction.

This research proposes a model-driven testing approach to generate executable test cases from visual use cases. The approach starts by describing the SUT requirements in use case models, which are transformed and mapped to abstract test scenarios. These scenarios are more refined and transformed into executable tests in a scripting language. The approach was applied and evaluated in an industrial 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).

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] 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

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

[8] 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.

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

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

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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

[16] Buhr, Raymond JA. "Use case maps as architectural entities for complex systems."Software Engineering, IEEE Transactions on 24.12 (1998): 1131-1155.

[17] Fowler, Martin. UML distilled: a brief guide to the standard object modeling language. Addison-Wesley Professional, 2004.

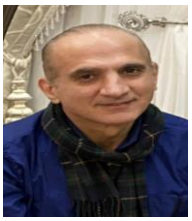
[18] ETSI ES 203 119 (stable draft): Methods for Testing and Specification (MTS); The Test Description Language (TDL).

[19] http://www.ttcn-3.org/index.php/downloads/standards.

[20] Adolph, S., Cockburn, A., & Bramble, P. (2002). Patterns for effective use cases. Addison-Wesley Longman Publishing Co., Inc.

[21] http://istar.rwth-aachen.de/tiki-index.php?page=jUCMNav.

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.