Automated test case generation from domain specific models of high-level requirements

Oyindamola Olajubu
University of Northampton, UK
oyindamola.olajubu@northampton.ac.uk

Suraj Ajit
University of Northampton, UK
suraj.ajit@northampton.ac.uk

Mark Johnson
University of Northampton, UK
mark.johnson@northampton.ac.uk

Scott Turner
University of Northampton, UK
scott.turner@northampton.ac.uk

Scott Thomson
GE Aviation, Cheltenham, UK
scott1.thomson@ge.com

Mark Edwards
GE Aviation, Cheltenham, UK
mark.edwards4@ge.com

ABSTRACT

Model-based software development has been shown to improve productivity and quality of software through automation. This involves using abstractions or models at several stages of development. This work reports on preliminary attempts to automate the generation of test cases from software requirement models using an industrial case study. The requirements are represented using a modeling notation and test cases are automatically generated using model to text transformation techniques.

CCS Concepts

• Software verification and validation →Software testing and debugging.

Keywords

Model-Based Testing, Domain Specific Languages

1. INTRODUCTION

Software Testing is one of the most crucial phases of development that could account for more than 50% of the overall cost of development [1]. The automation of this process could reduce this cost in terms of time and effort. Validation approaches such as requirement-based testing can be used to uncover faults and defects in artefacts during early stage development. To aid automated testing from requirements, software requirement specifications need to be precise. In this work, we propose an approach to automate test case generation from requirement models. This paper reports on an industrial case study. Requirement specification at GE Aviation Systems is primarily done using textual "shall" statements in natural language. Natural language is often used for requirement specifications due to its ease of understanding and the need for no additional training. Design models are then created based on these requirements for modeling and simulation. These models are then refined and used for automated code generation. The test cases developed against the requirements and design are developed manually. The natural language requirements are often ambiguous and manual testing can be time consuming and prone to human error. This work presents a requirement-based approach to automate the test case generation process to increase productivity. The requirements are represented as models and model-based approaches are applied to automatically generate test cases.

RACS'15, October 9–12, 2015, Prague, Czech Republic.

DOI: http://dx.doi.org/10.1145/2811411.2811555

Model-Based Software Development (MBSD) is an approach to software development with models as primary artefacts. Models or abstractions of the system at different levels are used at several phases of development. Model-Based Testing (MBT) involves the use of models in different formats as a basis for testing. In this work, our MBT approach uses requirements represented using a Domain Specific Language (DSL). DSLs are languages tailored to a particular domain or built for a specific purpose. They can be described as modeling languages whose constructs are based on domain related concepts. The use of DSLs have been shown to increase expressiveness of specifications by experts in a particular domain [2]. A DSL is used in this work to bridge the gap between ambiguous natural language specifications and rigorous formal specifications.

Automation of development activities in MBSD is usually done through a series of model transformations. Model transformations involve taking models conforming to a metamodel as input and generating development artefacts from them. With model transformations, models used at any stage of software development can be manipulated to generate a model or text used in another phase. There are two types of model transformations: Model-to-Model transformation (generation of models from other models) and Model-to-Text transformation (generation of textual artefacts from models). In this work, we apply Model-to-Text (M2T) transformation to generate text-based test cases from requirement models specified using the DSL. We present an approach to automate generation of test cases from textual requirement models expressed in a DSL. In this work, a M2T transformation language is applied for test case generation from requirement specifications expressed in a DSL. This paper is structured as follows: Section 2 presents some related work. The application of model transformations to test case generation is described in Section 3 while Section 4 concludes the paper and discusses future work.

2. RELATED WORK

There are several existing tools for automatic test case generation. However, the model-based approaches have been based on semi-

formal notations (UML, SysML) and formal specifications. Test cases have been generated from UML Activity diagrams [3][4][5][6], Use Cases [7][8], State Charts [9] and Sequence diagrams [10]. Our approach differs from these in that we use a textual modeling notation similar to natural language rather than a graphical notation. Although the modeling notation used for requirement specification is not fully described in this work, our work contrasts in its domain specificity compared to the genericity of the UML approaches.

Formal specifications have also been used for test case generation. Tools such as Fastest [11] and Isabelle [12] have been used to generate test cases from Z specifications. The authors of [13] describe an approach to test case generation from Object Z models. Formal models are mathematical-based notations for concise specification of systems. The use of a formal method requires a high learning curve while a DSL reduces it by the use of domain related concepts. The high learning curve of formal notations reduces its practical use in industry.

We take a less formal approach by using a DSL. It is expressive in its use of domain terms and can also be manipulated with existing model management tools for automatic generation of test cases. Cucumber [14] is a testing framework that allows domain experts (business managers) to specify tests in a plain language based on a behavior-driven development style of Given, When, Then, which any layperson can understand. These tests are then interpreted by Cucumber into the specified programming language. However, unlike our approach, it does not automate generation of test cases from high-level requirement specifications.

3. TEST CASE GENERATION

In this section, we describe the approach taken to automate the generation of test cases from textual specification models. The test cases generated are high-level artefacts that can be used to validate the resulting system against specified functional requirements. The generated tests are not intended to replace low-level unit tests. They are however aimed at automating the development of test descriptions that are otherwise manually written. The approach to test case generation is basically done using templates. M2T transformation is a model manipulation method for generating text from input models. Epsilon Generation Language (EGL) [15], a template based language for M2T transformation is used to develop EGL scripts for transforming testable requirements into test cases.

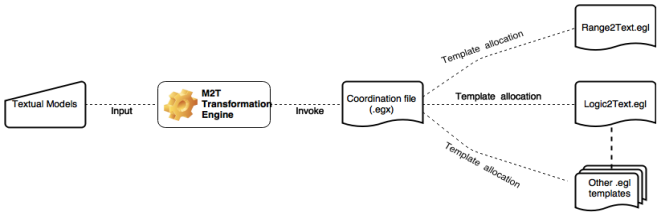

Figure 1: Overview of test case generation approach

An overview of the template based test case generation process is shown in Figure 1. The first step requires the textual models to be loaded for transformation. The textual models are requirements specified using a DSL, developed in collaboration with our industry partner. The description of the DSL is not described in this work and we have focused on the test case generation process. The DSL supports description of functional and non-functional requirements. The test cases generated are at a higher level compared to unit tests. Non-executable tests are generated from functional requirement models, assuming no knowledge of the implementation code. The test generation coverage of this approach is such that appropriate testing methods are applied to the different classes of requirements. The aim is to generate minimal effective tests rather than impractical exhaustive testing.

Table 1: Coordination file (.egx)

```
rule Logic2Text
transform logic: LogicRequirement{
template: "logic2text.egl"
target : "logic/" +logic.id+".txt"
}


rule Range2Text
transform range: RangeRequirement{
template: "range2text.egl"
target : "range/" +range.id+".txt"
}
```

The next phase includes coordination of the test generation by invoking model transformation file listed in Table 1. Within this coordination file, there are several rules specified for identified testable requirements. Logic requirements and Range requirements are examples of testable requirement types and are described in the following subsections. Corresponding folders for these requirement types are also created for population by generated text (.txt) files. The output of the test case generation for each requirement is written to the corresponding text file. This ensures traceability from each test case to its originating requirement. A test case is described in terms of a set of inputs and expected behavior or results.

As shown in the listing, the logic2text.egl template is applied to all identified logic requirements in the input specification model. A similar rule is defined for range requirements and can be extended for other types of requirements. In the example above, it is also specified that a 'range' folder is to be created and the range2text.egl template is applied to range requirements in the input model. The EGL scripts used for the test case generation have static and dynamic components. The dynamic components are populated based on the variable values of each requirement. Examples of these scripts are presented in the following subsections.

3.1 Range requirements

Range requirements are requirements that can be used to specify the upper and lower boundaries of a variable. It allows for the specification of a range of accepted integer values for a defined element or variable. Test case generation from range requirements is done by the application of equivalence partitioning and boundary testing. These types of requirements have defined upper and lower boundaries with optional margin values. The equivalence classes of range requirements are classified into lower boundary, upper boundary and derived midrange values. A total of nine test cases are generated for each requirement with three test cases for each class.

Table 2: Snippet from range2text.egl

```
if(self.range.margin.isDefined()){
margin =self.range.margin.marginvalue.asReal();
}else{ margin = 1.0;}
 %]
Lower Boundary Tests
Test case 1:[%=param%]=[%=self.getMin()- margin%]
Test case 2:[%=param%]=[%=self.getMin().asReal()%]
Test case 3:[%=param%]=[%=self.getMin()+ margin%]
```

shows a snippet from the range2text.egl transformation template. The 'if-else' condition assigns the value of 1.0 to the margin variable unless otherwise stated in the requirement. Three test cases are generated from the lower boundary of range requirements. The first test case subtracts the margin value from the lower boundary. The second test case is based on the lower boundary value itself and the third test case adds the margin value to the lower boundary value. These steps are repeated for the other equivalence classes, i.e., midrange and upper boundary values. The behavior of test cases within the specified range is expected to be normal while the system is expected to give some form of negative feedback with out of range test cases. The output for each requirement is a text based (.txt) file containing the test cases automatically generated from that requirement.

3.2 Logic requirements

Logic requirements are behavior requirements, which define a combination of one or more statements for elements or features. The format of logic requirements is such that a decision is made based on the output of the combination of multiple conditions and Boolean operators. A condition can be defined as an expression with no Boolean operators (e.g. **FT_DOOR = open** in the example in Figure 1). A decision is composed of conditions and zero or more Boolean operators. Logic operations could be evaluated as logic truth tables with each combination as a test case. This can however become exponential with increasing number of input conditions (i.e. 2^n, where n is the number of input conditions). An example of a logic requirement specified in the DSL is shown in Figure 2.

```
BREQ1: The TR_OPTION shall be set to true when
FT_DOOR = open AND LGT_ON= true AND PWR_SDBY =true
AND HTR_ON = false.
```

Figure 2: Example of logic requirement

The Modified Condition/Decision Coverage (MC/DC) criteria is applied as a guide to generating test cases from this type of requirements. This approach reduces the number of test cases by identifying combinations of conditions that will affect the overall decision. In [16], model checking was combined with MC/DC as a white-box testing criteria. In our work, we apply MC/DC as a black-box approach. The specifications used are models at a higher level of abstraction with no knowledge of the implementation code. The implementation of MC/DC is done based on the work presented in [17]. The walking true pattern is applied to requirements with only OR operators while the walking false pattern is applied to decisions based on only AND operators. The minimum number of test cases required to achieve MC/DC for a single operator logic based requirement is (n+1), where n is the number of conditions in the requirement.

Table 3: Logic table for logic-based requirement BREQ1

	FT_DOOR = open	LGT_ON = true	PWR_SDBY =true	HTR_ON = false	(Expected Output)
TC1	T (1,1)	T (1,2)	T (1,3)	T (1,4)	T
TC2	**F** (2,1)	T (2,2)	T (2,3)	T (2,4)	F
TC3	T (3,1)	**F** (3,2)	T (3,3)	T (3,4)	F
TC4	T (4,1)	T (4,2)	**F** (4,3)	T (4,4)	F
TC5	T (5,1)	T (5,2)	T (5,3)	**F** (5,4)	F

There are four conditions with only AND operators in the requirement in Figure 2. The total number of combinations and resulting test cases that could be generated is $2^4 = 8$, the application of "walkingFalse" identifies 4+1=5 effective test cases. To manually derive the test cases for this requirement, a logic table is used as shown in Table 3. The following are required to test decisions with only AND operators [16]:

- One test case where all inputs are true and expected output set to true. This is addressed in TC1 in Table 3
- N test cases, where each input is exclusively false and expected output set to false. Test cases (TC2 - TC5) show each condition in the requirement set to false exclusively.

Table 4: Code snippet of WalkingFalse operation in logic2text.egl

```
[%
operation LogicRequirement walkingFalse():Map{
%]
The conditions in this requirement are:
[% for (c in self.getConditions()){%]
[%=c%] [%}
var table:Map;
 var keys = self.getKeys();
'Populating table with all true
values.........'.println;
for (k in keys){
 table.put(k,true);
}
var tbrseq = self.toBeReplacedByFalse();
 for(d in tbrseq){
 table.put(d,false);
 }
 'Walking false complete.....'.println;
return table;  } %]
```

The egl script in Table 4 shows a method to automate the generation of the test cases. The conditions in the requirement are first identified and printed. A map is then used as a representation of the resulting logic table. The keys of the map are generated by the getKeys() operation. The keys are derived from the number of condition (i.e. 4) and the total number of test cases (i.e. 5). The keys for the map generated for this example are the values in brackets in Table 3. The generation of the map keys is followed by populating the map with true values. The keys of the map whose values is to be replaced by the walking false is then generated by the toBeReplacedbyFalse() method. This method is

used to calculate key values that produce the effect of a walking false as shown in Table 3. The result of the toBeReplacedbyFalse() method is a sequence of the following map keys: (2, 1) (3, 2) (4, 3) and (5, 4). The corresponding map values for these keys are then set to false. After the identified values have been replaced, the expected value of each row is calculated. Each row in the logic table is printed as a test case. The 'walkingTrue' pattern, applied to requirements with only OR operators is implemented in a similar manner. The map table is initially populated with false values and then the values of certain keys in the map are set to true.

4. CONCLUSION AND FUTURE WORK

This paper presents the work done to automate generation of test cases from domain specific models. A M2T transformation approach is applied to automate test case generation from requirement specifications in input models. A template-based approach is taken to generate test cases using EGL transformation scripts. Boundary testing with Equivalence classes is applied to range requirements with upper and lower boundary values. Test generated from single operator Logic Requirements was done using the MC/DC criteria. This is a work-in-progress paper and future work would involve extending the MC/DC approach to generate test cases from multiple operator logic requirements. This approach is to be extended to cover test case generation from other requirement types and also its application to larger models to evaluate its scalability.

5. ACKNOWLEDGMENTS

Many thanks to our industry partner GE Aviation and the University of Northampton for the joint sponsorship of this research project.

6. REFERENCES

[1] A. Pretschner, W. Prenninger, S. Wagner, C. Kuhnel, B. Sostawa, R. ZoIch, and T. Stauner, "One evaluation of model-based testing and its automation," in *Proceedings. 27th International Conference on Software Engineering, 2005. ICSE 2005.*, 2005, no. 2, pp. 392–401.

[2] A. Raja and D. Lakshmanan, "Domain Specific Languages," *International Journal of Computer Applications*, vol. 1, no. 21. pp. 105–111, 2010.

[3] D. Kundu and D. Samanta, "A Novel Approach to Generate Test Cases from UML Activity Diagrams.," *The Journal of Object Technology*, vol. 8, no. 3. p. 65, 2009.

[4] L. Wang, J. Yuan, X. Yu, J. Hu, X. Li, and G. Zheng, "Generating test cases from UML activity diagram based on gray-box method," in *Proceedings - Asia-Pacific Software Engineering Conference, APSEC*, 2004, no. 60233020, pp. 284–291.

[5] C. Mingsong, Q. Xiaokang, and L. Xuandong, "Automatic test case generation for UML activity diagrams," in *Proceedings of the 2006 international workshop on Automation of software test - AST '06*, 2006, p. 2.

[6] M. F. T. Boghdady, Pakinam N., Badr, Nagwa L., Hashem, Mohamed, and Tolba, "A Proposed Test Case Generation Technique Based on Activity Diagrams," *Int. J. Eng. Technol.*, vol. 11, no. 3, 2011.

[7] B. Hasling, H. Goetz, and K. Beetz, "Model Based Testing of System Requirements using UML Use Case Models," pp. 367–376, 2008.

[8] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic test generation: a use case driven approach," *IEEE Trans. Softw. Eng.*, vol. 32, no. 3, pp. 140–155, 2006.

[9] R. Swain, "Automatic Test case Generation From UML State Chart Diagram," vol. 42, no. 7, pp. 26–36, 2012.

[10] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado, "Test case generation by means of UML sequence diagrams and labeled transition systems," *2007 IEEE Int. Conf. Syst. Man Cybern.*, pp. 1292–1297, Oct. 2007.

[11] M. Cristiá, P. Albertengo, and P. Monetti, "Fastest: a model-based testing tool for the Z notation," in *Mazzanti, F., Trentani, G. (eds.) PTD-SEFM, Pisa, Italy, pp. 3-8 (2010)*.

[12] S. Helke, T. Neustupny, and T. Santen, "*Automating test case generation from Z specifications with Isabelle*." in ZUM '97 Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation, pp. 52-71, Springer-Verlang, 1997.

[13] A. Ashraf and A. Nadeem, "Automating the Generation of Test Cases from Object-Z Specifications," *30th Annu. Int. Comput. Softw. Appl. Conf.*, pp. 101–104, 2005.

[14] A. Bowers and J. Bell, "Automated testing with Selenium and Cucumber Write , batch , and run automated tests on your RIAs," , pp. 1–15, 2013,[online] Available at: http://www.ibm.com/developerworks/library/a-automating-ria/a-automating-ria-pdf.pdf [Accessed 15 Aug 2015].

[15] L. M. Rose, R. F. Paige, D. S. Kolovos, and F. A. C. Polack, "The epsilon generation language," in *Model Driven Architecture--Foundations and Applications*, 2008, pp. 1–16.

[16] S. Rayadurgam and M. P. E. Heimdahl, "Coverage based test-case generation using model checkers," *Proceedings. Eighth Annu. IEEE Int. Conf. Work. Eng. Comput. Based Syst. 2001*, pp. 83–91, 2001.

[17] K. J. Hayhurst, D. S. Veerhusen, J. J. Chilenski, and L. K. Rierson, *A Practical Tutorial on Modified Condition/Decision Coverage*. National Aeronautics and Space Administration, Langley Research Center, 2001.