Universidade Federal de Viçosa

EasyTest: An approach for automatic test cases generation from UML Activity Diagrams

Aluno: Fernando Augusto Diniz Teixeira

Orientadora: Gláucia Braga e Silva

Formato do trabalho: Artigo científico

Congresso de Submissão: 14th International Conference on Information Technology: New Generations (ITNG 2017)

Qualis: B1

Florestal

2016

EasyTest: An approach for automatic test cases generation from UML Activity Diagrams

Fernando Augusto Diniz Teixeira, Glaucia Braga e Silva
Institute of Technologics and Exact Sciences
Federal University of Viçosa - UFV
Florestal, Minas Gerais, Brazil
Email:{fernando.augusto,glaucia}@ufv.br

***Abstract*—The test cases generation is one of the great challenges for the Software Test Community because of the development efforts and costs to create, validate and test a large number of test cases. The automation of this process increases testing productivity and reduce labor hours. One technique that has been adopted to automate test cases generation is Model Based Testing (MBT). This paper proposes the EasyTest approach to generate test cases from UML Activity Diagrams aiming to integrate Modeling, Coding and Test stages in a software process and to reduce costs and development efforts. The proposed approach suggests an early detection of defects even in the modeling stage to prevent that unidentified defects are embedded in the coding stage. The work also presents the use of the generated test cases before and after the coding stage. To verify the proposed approach, this work also presents the EasyTest Tool to provide interoperability with the JUnit framework.**

***Keywords*-Test Automation; Model Based Testing; Gray-Box Testing; JUnit; TDD.**

1. INTRODUCTION

Empirical studies show that the test activity consumes a significant part of costs and development efforts and this expense becomes often cost-time prohibitive. The test cases generation represents a complex task in the test process because of the development effort and cost to create, validate and test a large number of test cases. To deal with this complexity some tools automate the generation of test cases from produced source code. Despite the benefits with automation, this technique still has problems to verify all system functionalities scenarios [1]. Furthermore, once a software defect is found the code must be fixed and verified again. This process causes rework, delays and increases the development costs.

Assuming that the earlier a defect is found the cheaper it is to fix it, the Model-Based Testing (MBT) technique appears to be promising. MBT consists of using various types of formal models to derive a set of test cases. MBT has a better performance than code-based approaches because it is a mixed approach of source code and requirements specification for testing the software [2]. Furthermore, MBT is more promising in terms of cost because the generated test cases can be used as starting point for the construction of a defect-free code. Unified Modeling Language (UML) models have long been used with the MBT technique because they have a lot of relevant information about system specification for test case design [3].

We propose an approach called "EasyTest" to generate test cases from UML activity diagrams aiming to integrate Modeling, Coding and Test stages in a software process and to reduce costs and development efforts. The EasyTest approach suggests an early detection of defects even in the modeling stage to prevent that unidentified defects are embedded in the coding stage. The generated test cases can be used in two scenarios: before and after the coding stage. If no code was produced, the test cases generated from the UML activity diagrams can be used by developers to produce defect-free code with Test Driven Development (TDD). Otherwise, after the coding stage, the generated test cases can be executed to verify (according to model specifications) and validate (expected behavior of an operation) the produced code in a test execution tool. To verify and to automate the proposed approach this work also presents the EasyTest Tool that provides interoperability with the JUnit frame-

work.

The paper is organized as follows: In Section 2, some related works are discussed. The EasyTest approach is presented in Section 3. Section 4 addresses the EasyTest tool. Finally, in Section 5, we discuss some conclusion and future works.

2. Related Works

There are different techniques and tools proposed for the automated generation of test cases. Some tools generate test cases from source code, but as mentioned by Shamshiri et al. [1] some tools did not present good test results for all the required functionalities.

Several studies address the use of MBT technique to generate test cases from different types of design and specification models. Nebut et al. [4] propose a test case generation approach from UML Use Case models and emphasize problems with interpretation caused by the use of natural language. Some researches address the generation of test cases from UML Class models and Object Constraint Language (OCL) specifications [5], [6].

Some works present techniques for transforming UML activity diagrams into graphs to generate test cases [7], [8] and provide tools to validate their proposals. However, these studies did not show a way to integrate the technique with other stages of a software process.

Pakinam et al. [3] extend the work of Linzhang et al. [7] to provide an architecture for automated generation of test cases from UML Activity Diagrams although they don't present an implementation of the proposal. Jena et al. [2] propose an approach similar to the work of Pakinam et al. [3] which uses a genetic algorithm to reduce the number of test cases while keeping the same coverage.

Test case generation techniques are commonly evaluated regarding the coverage criterion [9]–[11]. This criterion is one of the key metrics for evaluating the techniques performance. Other metrics such as time, cost, effort and generation complexity are also considered in the quality evaluation of the generated test cases [12].

The set of generated test cases can be used before the coding stage to produce defect-free code using the Test-Driven Development (TDD) strategy. These test cases provide a meaningful representation of alternative flows and branch conditions, enabling an easier application of the TDD for programmers. Latorre [13] shows how Unit Test-Driven Development (UTDD), a TDD subcategory, has a good performance in learning and application with programmers of different skill levels. Janzen and Saiedian [14] highlight how automated tools for unit tests have assisted improvements of TDD itself.

Based on the work of Linzhang et al. [7], the proposed approach consists of a gray-box test technique. This technique deals with problems which used to be ignored by both black-box and white-box techniques. It extends the logical coverage criteria of white box technique and finds all the possible paths from the model which describes the expected behavior of an operation. Then it generates test cases which can satisfy the path conditions expected by black-box technique.

3. EasyTest Approach

This section presents an automatic approach to generate test cases from UML activity diagrams using gray-box technique. The EasyTest approach comprises three phases, as shown in Figure 1: 1) importing activity diagrams in XMI; 2) test cases generation; and 3) applying test cases.

3.1. Phase 1 - Importing activity diagrams in XMI

Phase 1 aims to obtain and extract relevant information, about test case generation process, from XMI activity diagrams and then provide it to Phase 2. These information comprise properties of activities (vertices) and flows (edges). XMI (XML metadata Interchange) standard is defined by OMG (Object Management Group) and is widely used for exchanging metadata between UML modeling tools. Phase 1 uses activity diagrams drawn in different UML modeling tools as long as they have been exported to XMI according to OMG specification.

The EasyTest approach proposes interoperability with different types of UML modeling tools, importing XMI files in order to reuse activity diagrams previously drawn to avoid rework. This strategy is cheaper in terms of effort and time than those applied in other works that require the manual design of activity diagrams in their tools [7], [8].

To illustrate the use of EasyTest approach in a real case of development process, we adopt a case study

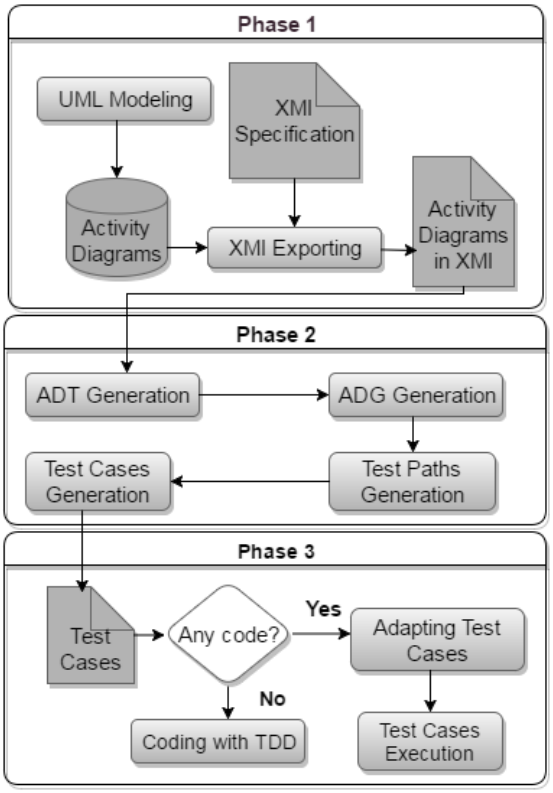

Fig. 1. EasyTest Approach

based on a Control System for *Aedes Aegypti* that is being developed in an academic interdisciplinary project at Federal University of Viçosa - campus Florestal. For Phase 1, an Activity Diagram was produced to represent an use case from this system that refers to an operation called ”Create Complaint” used by citizens to report an occurrence of *Aedes Aegypti* vector in an infested area.

The activity diagram mentioned above is illustrated in Figure 2. This diagram was produced in Oracle JDeveloper 12c tool (version 12.1.3.0.0) and exported to XMI format.

It is important to note that some edges in the diagram contain labels with textual stereotypes representing data flow for that edge. For example, the edge ”String::Reference Point” contains the variable name ”Reference Point” of the type ”String” from the activity ”Get Reference Point” to ”Get Description”. Values like that are relevant for the successful generation of test cases in Phase 2.

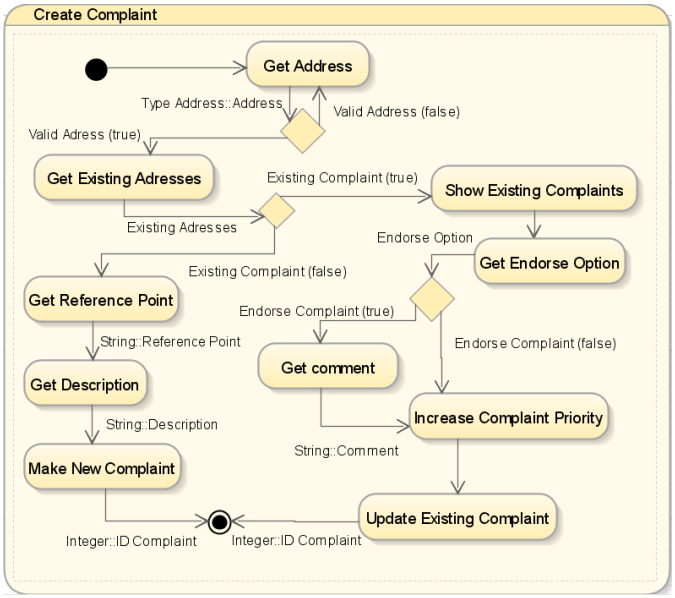

Fig. 2. Activity Diagram for the operation ”Create Complaint”

3.2. Phase 2 - Test Cases Generation

Phase 2 is responsible for processing the XMI Activity Diagram to find Test Paths and then generate the final test cases. This phase was based on some techniques proposed by [2], [3], [15].

The development of this phase is comprised by four steps that are presented below as well as the applied improvements proposed in this work.

3.2.1. ***Activity Dependency Table (ADT) Generation****:* The first step of this phase is the ADT generation to present inputs, outputs and dependencies of all activity diagram elements. Figure 3 shows the generated ADT for the Activity Diagram (Figure 2).

The elements are described at each row of the table. The table columns present information about element name, inputs, expected outputs, dependencies relationships between elements and edges values in some cases. The elements containing the term ”Validate” in the ADT correspond to the Decision Nodes of the Activity Diagram.

3.2.2. ***Activity Dependency Graph (ADG) Generation****:* After the ADT generation the next step is the ADG generation. The symbols in ADT become vertices in ADG and dependencies of each symbol become edges between the corresponding vertices. The ADG generation is a significant part of the EasyTest approach to gather test paths corresponding to the activity diagram.

ID	Element Name	Dependency	Input	Expected Output
0	Initial Node			
1	Get Address	0		Address
		2	Valid Address(false)	Address
2	Validate Address	1	Address	Valid Address(true)
		1	Address	Valid Address(false)
3	Get Existing Adresses	2	Valid Address(true)	
4	Validate Existing Adresses	3		Existing Complaint(true)
		3		Existing Complaint(false)
5	Show existing Complaints	4	Existing Complaint(true)	
6	Get Reference Point	4	Existing Complaint(false)	Reference Point
7	Get Endorse Option	5		Endorse Option
8	Get Description	6	Reference Point	Description
9	Validate Endorse Option	7	Endorse Option	Endorse Complaint(false)
		7	Endorse Option	Endorse Complaint(true)
10	Make New Complaint	8	Description	ID Complaint
11	Increase Complaint Priority	9	Endorse Complaint(false)	
		12	Comment	
12	Get comment	9	Endorse Complaint(true)	Comment
13	Activity Final Node	14	ID Complaint	
		10	ID Complaint	
14	Update existing complaint	11		ID Complaint

Fig. 3. Generated ADT

Figure 4 shows the generated ADG for the ADT (Figure 3).

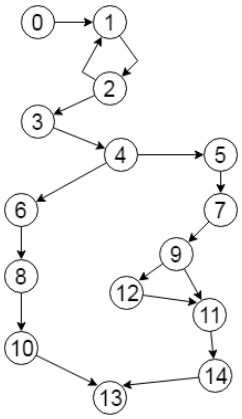

Fig. 4. Generated ADG

3.2.3. ***Test Paths Generation****:* For the generation of Test Paths from the ADG generated previously, we developed an algorithm based on the depth-first search for graphs. The final set of Test Cases are defined from the resulting test paths.

The strategy used to generate these Test Paths impacts directly the test cases quality. The algorithm was built aiming to satisfy the criterion ”Transition Coverage”. As defined in [9] and [16], Transition Coverage intends to verify all transitions in the activity diagram. The transition coverage result is a ratio between the verified transitions and all Activity Diagram transitions. This criterion guarantees that each loop condition is checked and an iteration is executed at least one time. By this way, we also guarantee that other coverage criteria as ”Activity Coverage” and ”Branch Coverage” are satisfied. The ”Activity Coverage” intends to verify all Activity States in the Activity Diagram. The ”Branch Coverage” intends to verify all conditions branches in the Activity Diagram so that all edges are checked.

Based on the ”Transition Coverage” and using the generated ADG (Figure 4) the following Test Paths were obtained:

Test Path 1: $0, 1, 2, 3, 4, 5, 7, 9, 11, 14, 13$
Test Path 2: $0, 1, 2, 3, 4, 5, 7, 9, 12, 11, 14, 13$
Test Path 3: $0, 1, 2, 3, 4, 6, 8, 10, 13$
Test Path 4: $0, 1, 2, 1, 2, 3, 4, 5, 7, 9, 11, 14, 13$

Another criterion that could be considered is ”All Activity Path Coverage”. This criterion intends to verify all different activity sequences from the Initial Node to the Final Node. Considering this criterion for the ADG shown in Figure 4 we would get six Test Paths. The choice of the ”Transition Coverage” was based on the fact that if the loop is executed at least one time and a Final Node is reached, other tests for the same loop condition are not required. Thereby, the ”Transition Coverage” can provide the same coverage with fewer test cases.

3.2.4. ***Test Cases Generation****:* The last step of Phase 2 is the test cases generation from the ADT and Test Paths. According to the number of Test Paths, this step generated four Test Cases for the given Example. Figure 5 shows one of these generated test cases in Phase 2 of the EasyTest approach. It can be observed that all input and output values of each path vertice are detailed as well as the input and output values of the test case.

3.3. Phase 3 - Applying Test Cases

Phase 3 has the purpose of applying the generated test cases in the test execution and coding

Test Case Number	Test Path Vertice ID	Vertice Input	Vertice Output	Test Case Input	Test Case Expected Output
1	0			Address::Type Address @Address !=null; Endorse Option:: Boolean @Endorse Option ==false;	ID Complaint:: Integer;
	1		Address		
	2	Address	Valid Address(true)		
	3	Valid Address(true)			
	4		Existing Complaint(true)		
	5	Existing Complaint(true)			
	7		Endorse Option		
	9	Endorse Option	Endorse Complaint(false)		
	11	Endorse Complaint(false)			
	14		ID Complaint		
	13	ID Complaint			

Fig. 5. Example of a Generated Test Case

stages. In the EasyTest approach, the resulting test cases can be used to verify or produce code in any programming languages. Different Test Cases Execution tools can be used in this approach as long as a mapping has been designed to adapt the test cases for the code structure used in the selected tool.

Phase 3 comprises two usage scenarios within the software development process: before and after the coding stage. The possibility of application in different scenarios allows a better adherence of the testing activity in the software development process. The two different scenarios are discussed bellow.

3.3.1. ***Before Coding Stage - Test Driven Development (TDD)****:* In this scenario, TDD is applied to build defect-free code. Once the test cases have been generated, in addition to a meaningful representation of alternative flows and branch conditions, the process of developing code becomes much simpler because the code structure will be guided by these resources. As shown by [13], When test driven development is applied with unit tests, we have a good performance in learning and application with programmers of different skill levels.

Although in the TDD context models are not usually created, the EasyTest approach can provide advantages in terms of time and effort because of the automatic generation of test cases. In addition, the approach can help programmers to produce defect-free code from the beginning of the development software process because they can use test cases to verify all functionalities and their flows, thus compensating for the time required to develop the activity diagrams.

To verify this scenario, the authors made a quick single experiment to develop a defect-free code applying TDD for the operation ”Create Complaint”. For the experiment, the code was produced based on test cases generated by the approach and the activity diagram (Figure 2). From these resources, it was possible to quickly identify conditional and iterative blocks as well as the possible values for each flow.

For the experiment, we used EClEmma, a free Java code coverage tool. This tool was used to evaluate the line coverage of the produced code and a coverage of nearly 92% was achieved. This result highlights the quality of the generated test cases and how much the code is compatible with the activity diagram. However, other experiments will be made to check the results for different contexts.

3.3.2. ***After Coding Stage - Exporting generated Test Cases****:* In this scenario, we have an activity diagram and a code made for it. The source code is used to verify if it matches with the results of the test cases execution, that is, if the code execution logic is consistent with the activity diagram expected behavior. Otherwise, it is necessary to evaluate the changes required in the code to satisfy the modeling specifications. As long as the code is compatible with the model we have a valid test for all code functionalities and flows.

Also, we have an automatic generation of test code that comprises the generated test cases for execution in any Test Execution Tool. To achieve this, it is required a mapping between the generated test cases and the technical structure of the selected tool.

Once the activity diagram is source code independent, we assign generic names to classes and operations which will later be adjusted by testers/programmers.

4. EASYTEST TOOL

To verify the EasyTest approach, we developed a Java tool called ”EasyTest tool” that provides automatic support for the Phases 2 and 3. According to the Phase 1 of the approach, EasyTest Tool uses activity Diagrams represented in XMI files.

This tool version provides, in step "Adapting Test Cases", test cases for Java through a mapping for integration with the JUnit framework.

The EasyTest tool provides an interface where for each generated test cases, specific instructions and fields are displayed so the tester/programmer can insert the required test data. In addition, the interface displays a dynamic graph highlighting corresponding Test Paths for each Test Case to support the filling of fields by the user. With this resources the tool can generate a code for JUnit capable to test all functionalities and flows, reducing efforts and time of creating test cases and increasing quality of test cases in spite programmers of different skill levels.

5. Conclusion and Future Work

This work presented the EasyTest approach of automatic test cases generation from UML activity diagrams and addressed the use of these test cases in two scenarios: before and after the coding stage. For the first scenario, the approach supplies TDD programmers with relevant information about the functionality logic in order to ease the process of developing a defect-free code. For the second scenario, the approach proposes an automatic generation of test code that comprises the test cases to execute them in any Test Execution Tool. To verify this scenario, the EasyTest tool was developed to provide interoperability with JUnit.

The EasyTest is based on gray-box technique, allowing the identification of problems which used to be ignored by both black-box and white-box techniques as it extends the logical coverage criteria and finds all the paths from the designed model, which describes the expected behavior of an operation.

Based on some experiments and observations, the EasyTest approach can provide some gains in terms of cost, effort and time. This gains can be observed in contexts such as: reuse of activity diagrams previously designed; earlier detection of defects before coding stage; support for the defect-free code development; and increase in the test case quality because the use of a gray-box technique.

For future works, we intend to make more experiments for the two scenarios of the EasyTest Phase 3, involving a set of senior testers/programmers to evaluate and refine the EasyTest approach.

References

[1] S. Shamshiri, R. Just, J. M. Rojas, G. Fraser, P. McMinn, and A. Arcuri, "Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges (t)," in *Automated Software Engineering (ASE), 2015 30th IEEE/ACM International Conference on*. IEEE, 2015, pp. 201–211.

[2] A. K. Jena, S. K. Swain, and D. P. Mohapatra, "A novel approach for test case generation from uml activity diagram," in *Issues and Challenges in Intelligent Computing Techniques (ICICT), 2014 International Conference on*. IEEE, 2014, pp. 621–629.

[3] P. N. Boghdady, N. L. Badr, M. Hashem, and M. F. Tolba, "A proposed test case generation technique based on activity diagrams," *International Journal of Engineering & Technology IJET-IJENS*, vol. 11, no. 03, 2011.

[4] C. Nebut, F. Fleurey, Y. Le Traon, and J.-M. Jezequel, "Automatic test generation: A use case driven approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 3, pp. 140–155, 2006.

[5] C.-K. Chang and N.-W. Lin, "Utgen: A black-box method-level unit-test generator for junit test-platform," in *Trustworthy Systems and Their Applications (TSA), 2015 Second International Conference on*. IEEE, 2015, pp. 1–7.

[6] F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting, "A subset of precise uml for model-based testing," in *Proceedings of the 3rd international workshop on Advances in model-based testing*. ACM, 2007, pp. 95–104.

[7] W. Linzhang, Y. Jiesong, Y. Xiaofeng, H. Jun, L. Xuandong, and Z. Guoliang, "Generating test cases from uml activity diagram based on gray-box method," in *Software Engineering Conference, 2004. 11th Asia-Pacific*. IEEE, 2004, pp. 284–291.

[8] 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*. ACM, 2006, pp. 2–8.

[9] M. Chen, P. Mishra, and D. Kalita, "Coverage-driven automatic test generation for uml activity diagrams," in *Proceedings of the 18th ACM Great Lakes symposium on VLSI*. ACM, 2008, pp. 139–142.

[10] M. Chen, X. Qiu, W. Xu, L. Wang, J. Zhao, and X. Li, "Uml activity diagram-based automatic test case generation for java programs," *The Computer Journal*, vol. 52, no. 5, pp. 545–556, 2009.

[11] J. A. McQuillan and J. F. Power, "A survey of uml-based coverage criteria for software testing," *Department of Computer Science. NUI Maynooth, Co. Kildare, Ireland*, 2005.

[12] P. B. Nirpal and K. Kale, "A brief overview of software testing metrics," *International Journal on Computer Science and Engineering (IJCSE)*, vol. 3, no. 1, pp. 204–211, 2011.

[13] R. Latorre, "Effects of developer experience on learning and applying unit test-driven development," *IEEE Transactions on Software Engineering*, vol. 40, no. 4, pp. 381–395, 2014.

[14] D. S. Janzen and H. Saiedian, "Test-driven development: Concepts, taxonomy, and future direction," *Computer*, vol. 38, no. 9, pp. 43–50, 2005.

[15] P. N. Boghdady, N. L. Badr, M. A. Hashim, and M. F. Tolba, "An enhanced test case generation technique based on activity diagrams," in *Computer Engineering & Systems (ICCES), 2011 International Conference on*. IEEE, 2011, pp. 289–294.

[16] R. K. Swain, V. Panthi, and P. K. Beher, "Generation of test cases using activity diagram," *International journal of computer science and informatics*, vol. 2, no. 2, pp. 2231–5292, 2013.