Decomposition of Test Cases in Model-Based Testing

Marcel Ibe

Clausthal University of Technology
Clausthal-Zellerfeld, Germany
marcel.ibe@tu-clausthal.de

Abstract. For decades software testing is a fundamental part in software development. In recent years, model-based testing is becoming more and more important. Model-based testing approaches enable the automatic generation of test cases from models of the system to build. But manually derived test cases are still more efficient in finding failures. To reduce the effort but also keep the advantages of manually derived test cases a decomposition of test cases is introduced. This decomposition has to be adapted to the decomposition of the system model. The objective of my PhD thesis is to analyse these decompositions and develop a method to transfer them to the test cases. That allows the reusing of manually derived test cases at different phases of a software development project.

Keywords: model-based testing; model decomposition; test case decomposition

1 Introduction

During a software development project, testing is one of the most important activities to ensure the quality of a software system. About 30 to 60 per cent of the total effort within a project is spent for testing [19], [14]. This value did not change during the last three decades. Even though testing is a key aspect of research and there are constantly improving methods and tools that can be applied. One fundamental problem during testing is the fact, that it is not possible to show completely absence of errors in a software system.
[7]. Nevertheless by executing enough test cases a certain level of correctness can be ensured. But the number of test cases must not be too large otherwise it would not be efficient to test the systems any longer. One of the most important challenges is to create a good set of test cases. That means the number of test cases should be minimal but it should also test as much as possible of the systems behaviour.
Model-based testing is one technique that addresses this problem. A potential infinite set of test cases is generated from the test model, an abstract model of the system to construct. Based on a test case specification a finite set of these generated test cases can be selected [16]. These test cases can be executed manually or automatically.

2 Related Work

In [20] a distinction is made between four levels of testing: acceptance testing, system testing, integration testing and component or unit testing. The focus here is integration testing and unit testing. Furthermore testing approaches can be divided by the kind of model from which the test cases are generated [6].
Tretmans describes an approach to generate test cases from labelled transition systems [17]. He introduces the ioco-testing theory. By this theory it is possible to define for example when a test case has passed. An algorithm is introduced that allows generating test cases from labelled transition systems. Several tools implement the ioco-testing theory. For example TorX [18], TestGen [10] or the Agedis Tool [9].
Jaffuel and Legeard presented an approach in [12] that generates test cases for functional testing. The test model is described by the B-notation [1]. Different coverage criteria allow the selection of test cases.
Another approach was described in [13] by Katara and Kervinen. It bases on so called action machines and refinement machines. These are also labelled transitions systems with keywords as labels. Keyword based scenarios are defined by use cases. Then they are mapped to the action machines and detailed by the refinement machines.
An approach that generates test cases for service oriented software systems from activity diagrams is introduced in [8]. Test stories are derived from activity diagrams. These user stories are the basis for generating test code. Several coverage criteria can be checked by constraints.
In [15] Ogata and Matsuura describe an approach that is also based on activity diagrams. It allows the creation of test cases for integration testing. Use cases from use case diagrams are refined by activity diagrams. For every system or component that is involved in an activity diagram there is an own partition. So it is possible to select only these actions from the diagram, which define the interface between the systems or the components. Now the test cases can be generated from these actions.
Blech et al. describe an approach in [4] which allows the reusing of test cases in different levels of abstractions. For that purpose relations between more abstract and more concrete models are introduced. After that they try to prove, that the more concrete model is in fact a refinement of the more abstract model. That approach is based on the work of Aichernig [2]. He used the refinement calculus from Back and von Wright [3] to create test cases from requirements specifications by abstraction.
In [5] Briand et al. introduce an approach to make an impact analyse. So it is possible to determine which test cases are affected by changes of the model. The test cases are divided into different categories and can be handled respectively. That approach was developed to determine the reusability of test cases for regression testing during changes within the specification.

3 Problem

Nowadays there are a lot of approaches that are able to generate test cases for different kinds of tests from a model of a system automatically. The advantage of these approaches is that the effort for the test case generation is low. Furthermore a test specification can ensure that the test cases meet several criteria of test coverage. What is not considered there is the efficiency of the test cases. That means a set of generated test cases can more or less test the whole system. But therefor maybe a huge number of test cases is necessary. However, manually derived test cases can be much more efficient. Because of its experience a test architect is able to derive such test cases that test the most error-prone parts of a system. So a smaller set of test cases can cover a big and important part of the system. But such a manual test case derivation is more expensive than an automatic generation. This addition effort cannot be balanced by the less number of test cases that has to be executed.
During a software development project there are different kinds of tests that test the system or parts of it. For this purpose for every kind of test there are new test cases necessary. Starting with the creation of test cases for system testing from the requirements to creation of test cases for integration and unit testing from the architecture, at every test case creation there is the possibility to choose the manual test case derivation or the automatic test case generation with all its advantages and disadvantages. The more complex the model of the system gets the more the automatic generation is in advantage over the manual derivation because there is a point where a model is not manageable for a person any longer. Generally the requirements model is much smaller than the architecture because the latter contains much more additional information about the inner structure and behaviour of the system. Therefore, a manual test case derivation is more reasonable for system testing than for integration or unit testing. But the advantages of the manually derived test cases are limited to system testing. The approach that is introduced in the following section should automatically transfer the advantages of manually derived test cases for system testing to test cases for integration and unit testing. This is done by decomposing the test cases. In this way the information that were added to the test cases during derivation can be reused for further test cases but without the effort for another manual test case derivation.
The question that should be answered in the doctoral thesis is: Can the advantages of manually derived test cases over automatically generated ones be transferred to another level of abstraction by an automatic decomposition of these test cases?

4 Proposed Solution

To use the advantages of manually derived test cases for at least one time in the project a set of test cases has to be derived manual. As stated above, suitable for this are the test cases for system testing. They can be derived from the requirements model. It does not contain details about the system like the internal structure or behaviour. So the test architect can aim solely at the functions of the complete system. Hence one can get a set of test cases to test the system against its requirements. Based on the requirements an architecture of the system is created after that. This architecture is getting more and more refined and decomposed. For example the system itself can be decomposed into several components that can be decomposed into subcomponents again. The functions can be decomposed analogue into subfunctions which are provided by the components. To test the particular subfunctions and the interaction of the components integration and unit test are executed. Therefore, test cases are required again. They could be derived from the architecture. But that would entail much additional effort. Another option is an automatic generation. But that would mean to lose the advantages of manually derived test cases. A third option is to reuse the manually derived test cases from system testing. To do this the following problem has to be solved. Meanwhile there are additional information added to the architecture for example information about the internal composition or subfunctions of the system. The test cases also need these information. For instance it is not possible to test a function if there is no test case that has information about the existence of that function. Hence, the refinements and decompositions that were made at the architecture must also be made at the test cases. That means the test cases also has to be decomposed. After that the test cases from system testing can be used as basis for test cases for integration and unit testing. A manual re-deriving of test cases is not necessary any longer. Figure 1 shows this process schematically.

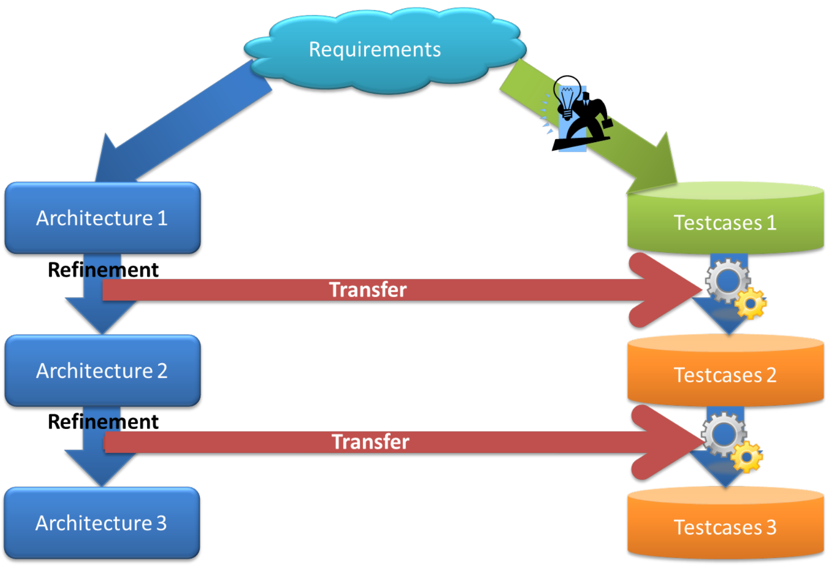
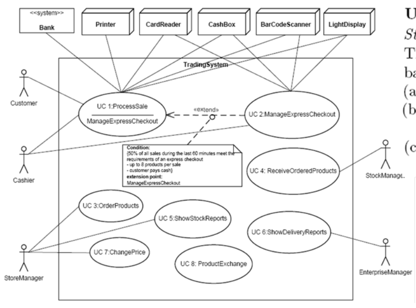
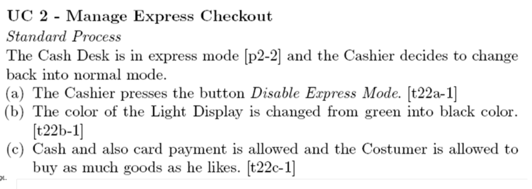


Fig. 1. Creation of test cases during project

To illustrate how such a decomposition of test cases could look like it is shown at the Common Component Modelling Example (CoCoME) [11]. CoCoME is the component based model of a trading system of a supermarket. Here we focus only on the CashDesk component of the trading system. The requirements model contains the trading system itself and other systems and actors of its environment. Besides the models that describe the static structure of the system the behaviour is described by usecases. Such a usecase is for example the handling of the express mode of the cash desk. Under certain conditions a cash desk can switch into the express mode. That means that a customer can buy a maximum of eight products at that cash desk and has to pay cash. Card payment is not allowed any longer. The cashier can always switch off the express mode at his cash desk. Figur 2 shows the system environment and an excerpt of the usecase *Manage Express Checkout*. A test case that tests the management of the express mode could consists of the follwoing three steps:

1. The cashier presses the button *DisableExpressMode* at his cash desk.
2. The cash desk ensures that the colour of the light display changes to black.
3. The cash desk ensures that the card reader accepts credit card again and card payment is allowed.





UC 2 - Manage Express Checkout

Standard Process

The Cash Desk is in express mode [p2-2] and the Cashier decides to change back into normal mode.

(a) The Cashier presses the button *Disable Express Mode.* [t22a-1]
(b) The color of the Light Display is changed from green into black color. [t22b-1]
(c) Cash and also card payment is allowed and the Costumer is allowed to buy as much goods as he likes. [t22c-1]

Fig. 2. Example of the system environment and one usecase

In the next step the complete system is decomposed into several components. One of these components is the *CashDeskLine* that also contains a set of *CashDesk* sub components. Also the description of the behaviour, in this case the management of the express mode, is decomposed into smaller steps (see figure 3).

Similarly, the test case that was defined above has to be decomposed to test the new subfunctions. After the decomposition it would look as follows:

1. The cashier presses the button *DisableExpressMode* at his cash desk.
   (a) The cashier presses the button *DisableExpressMode* th his cash box.
   (b) The cash box sends an *ExpressModeDisableEvent* to the cash box controller.

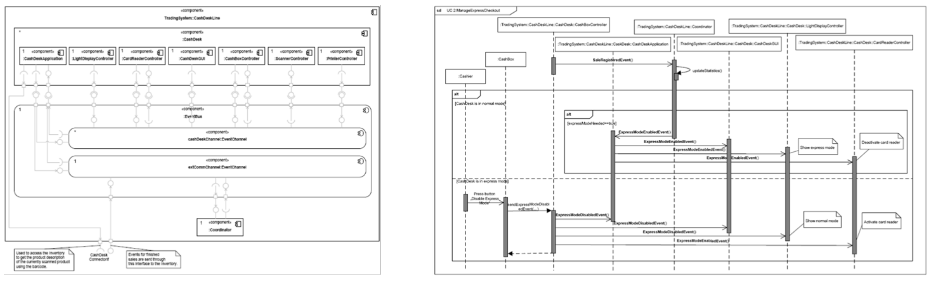

«component»
TradingSystem::CashDeskLine
«component»
:CashDesk
«component»
:CashDeskApplication
«component»
:LightDisplayController
«component»
:CardReaderController
«component»
:CashDeskGUI
«component»
:CashBoxController
«component»
:ScannerController
«component»
:PrinterController
«component»
:EventBus
«component»
cashDeskChannel:EventChannel
«component»
extCommChannel:EventChannel
«component»
:Coordinator
Used to access the inventory to get the product description of the currently scanned product using the barcode.
CashDesk ConnectorIf
Events for finished sales are sent through this interface to the inventory.
sd UC 2:ManageExpressCheckout
:TradingSystem::CashDeskLine::CashDesk::CashBoxController
:TradingSystem::CashDeskLine::Coordinator
:TradingSystem::CashDeskLine::CashDesk::LightDisplayController
:TradingSystem::CashDeskLine::CaskDesk::CashDeskApplication
:TradingSystem::CashDeskLine::CashDesk::CashDeskGUI
:TradingSystem::CashDeskLine::CashDesk::CardReaderController
:CashBox
:Cashier
SaleRegisteredEvent()
updateStatistics()
alt
[CashDesk is in normal mode]
alt
[expressModeNeeded==true]
ExpressModeEnabledEvent()
ExpressModeEnabledEvent()
ExpressModeEnabledEvent()
ExpressModeEnabledEvent()
Show express mode
Deactivate card reader
[CashDesk is in express mode]
Press button „Disable Express Mode"
sendExpressModeDisabledEvent(...)
ExpressModeDisabledEvent()
ExpressModeDisabledEvent()
ExpressModeDisabledEvent()
ExpressModeEnabledEvent()
Show normal mode
Activate card reader

Fig. 3. Example of the decomposed structure of the systems and its behaviour

   (c) The cash box controller sends an *ExpressModeDisableEvent* to the cash desk application.
2. The cash desk ensures that the colour of the light display changes to black.
   (a) The cash desk application send an *ExpressModeDisabledEvent* to the light display controller.
3. The cash desk ensures that the card reader accepts credit card again and card payment is allowed.
   (a) The cash desk application sends an *ExpressModeDisabledEvent* to the card reader controller.

The steps at the ordinary level are identic to that from the original test case. Because of the information about the additional components and the communication between them, there are a few new test steps at the second level necessary. New these new components and their communication can also be tested by this test case. So the test case for system testing that was created from the requirements can also be used for integration and unit testing.
To do that test case decomposition the following challenges has to be addressed:

– Definition of associations between requirements or the first architecture and the manually derive test cases. This is necessary to transfer the decompositions that are made at the architecture to the test cases.
– Tracing of the elements of the architecture during the further development. So it can be decided which elements of the requirements or architecture are decomposed.
– Definition of the decomposition of the test cases. Now that it has been established how the elements of the architecture are decomposed and the corresponding test cases can be identified it can be analysed how the test cases has to be decomposed according to the decomposition of the architecture elements.
– Automatic transfer of the decomposition steps from the architecture to the test cases. Therefore all the possible decomposition steps have to be analysed and classified. After that they can be detected automatically and the corresponding test cases can be decomposed respectively.

5 Contribution, Evaluation and Validation

The objective of this PhD thesis is to develop an approach for decomposing test case analogous to the decomposition of the corresponding system to test. That approach is based upon findings about the decomposition of a system influences the corresponding test cases. In another step the approach shall be implemented as a prototype. After that the prototype can be evaluated and compared to other implementations of model-based testing approaches.
Within the next year the decomposition steps of a system and their influence to the corresponding test cases shall be analysed. For this the changes of individual model elements during detailed design have to be traced. Especially how they are extended with information about their interior structure and behaviour. Another important fact is the relation between model elements and test steps. With this knowledge it is possible to adapt the test cases after a decomposition of the system in a way that the test cases can cover also the added information about structure and behaviour.
In the six following months a first prototype shall be implemented. It is intended to evaluate this prototype within a student project. To see how efficient the test cases are that were derived with this approach a set of manually derived test cases are compared with a set of automatically generated ones. After this the manually derived test cases are decomposed. In the next step the decomposed test cases are compared with new automatically generated test cases. In each case the average number of failures that are detected by the test cases and how serious these failures for the function of the system are compared. The findings from this first evaluation are integrated in the approach and the prototype during the next six months. After that finalisation the new prototype should be set up in an industrial project and compared with other model-based tools in use.

References

1. Abrial, J.R., Abrial, J.R.: The B-Book: Assigning Programs to Meanings. Cambridge University Press (Nov 2005)
2. Aichernig, B.K.: Test-design through abstraction - a systematic approach based on the refinement calculus. j-jucs 7(8), 710 – 735 (Aug 2001)
3. Back, R.J.R.: Refinement Calculus: A Systematic Introduction. Springer (Jan 1998)
4. Blech, J.O., Mou, D., Ratiu, D.: Reusing test-cases on different levels of abstraction in a model based development tool. arXiv e-print 1202.6119 (Feb 2012), `http://arxiv.org/abs/1202.6119`, EPTCS 80, 2012, pp. 13-27
5. Briand, L., Labiche, Y., Soccar, G.: Automating impact analysis and regression test selection based on UML designs. In: International Conference on Software Maintenance, 2002. Proceedings. pp. 252–261 (2002)
6. Dias Neto, A.C., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. In: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007. p. 3136. WEASELTech

'07, ACM, New York, NY, USA (2007), http://doi.acm.org/10.1145/1353673.1353681
7. Dijkstra, E.W.: The humble programmer. Commun. ACM 15(10), 859866 (Oct 1972), http://doi.acm.org/10.1145/355604.361591
8. Felderer, M., ChimiakOpoka, J., Breu, R.: Modeldriven system testing of service oriented systems. In: Proc. of the 9th International Conference on Quality Software (2009), http://www.dbs.ifi.lmu.de/~fiedler/publication/FZFCB09.pdf
9. Hartman, A., Nagin, K.: The AGEDIS tools for model based testing. In: Proceedings of the 2004 ACM SIGSOFT international symposium on Software testing and analysis. p. 129132. ISSTA '04, ACM, New York, NY, USA (2004), http://doi.acm.org/10.1145/1007512.1007529
10. He, J., Turner, K.J.: Protocol-inspired hardware testing. In: Csopaki, G., Dibuz, S., Tarnay, K. (eds.) Testing of Communicating Systems, pp. 131–147. No. 21 in IFIP The International Federation for Information Processing, Springer US (Jan 1999), http://link.springer.com/chapter/10.1007/978-0-387-35567-2_9
11. Herold, S., Klus, H., Welsch, Y., Deiters, C., Rausch, A., Reussner, R., Krogmann, K., Koziolek, H., Mirandola, R., Hummel, B.: CoCoME-the common component modeling example. In: The Common Component Modeling Example, p. 1653. Springer (2008), http://link.springer.com/chapter/10.1007/978-3-540-85289-6_3
12. Jaffuel, E., Legeard, B.: LEIRIOS test generator: automated test generation from b models. In: Proceedings of the 7th international conference on Formal Specification and Development in B. p. 277280. B'07, Springer-Verlag, Berlin, Heidelberg (2006), http://dx.doi.org/10.1007/11955757_29
13. Katara, M., Kervinen, A.: Making model-based testing more agile: A use case driven approach. In: Bin, E., Ziv, A., Ur, S. (eds.) Hardware and Software, Verification and Testing, pp. 219–234. No. 4383 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 2007), http://link.springer.com/chapter/10.1007/978-3-540-70889-6_17
14. Myers, G.J., , B., Thomas, T.M., Sandler, C.: The art of software testing. John Wiley & Sons, Hoboken, N.J. (2004)
15. Ogata, S., Matsuura, S.: A method of automatic integration test case generation from UML-based scenario. WSEAS Trans Inf Sci Appl 7(4), 598607 (2010), http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.175.5822&rep=rep1&type=pdf
16. Pretschner, A., Philipps, J.: 10 methodological issues in model-based testing. In: Broy, M., Jonsson, B., Katoen, J.P., Leucker, M., Pretschner, A. (eds.) Model-Based Testing of Reactive Systems, pp. 281–291. No. 3472 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 2005), http://link.springer.com/chapter/10.1007/11498490_13
17. Tretmans, J.: Model based testing with labelled transition systems. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) Formal Methods and Testing, pp. 1–38. No. 4949 in Lecture Notes in Computer Science, Springer Berlin Heidelberg (Jan 2008), http://link.springer.com/chapter/10.1007/978-3-540-78917-8_1
18. Tretmans, J., Brinksma, E.: TorX: automated model-based testing. pp. 31–43. Nuremberg, Germany (Dec 2003), http://doc.utwente.nl/66990/
19. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann (Jul 2010)
20. van Veenendaal, E.: Standard glossary of terms used in software testing. International Software Testing Qualifications Board (2010)