Author 1, Author n

Title

Technical Report

Advisor University of Heidelberg
Prof. Dr. Barbara Paech, Astrid Rohmann

mm dd, year

Abstract

Place the abstract in this section.

Contents

1 Introduction

Lorem ipsum dolor sit amet, consectetuer adipiscing **Figure 1.1** elit. Vivamus elementum sem eget tortor. Pellentesque id orci cursus sem tempus porttitor. Aenean tincidunt, neque vitae bibendum lacinia, magna erat dapibus nunc, vel pharetra nibh erat ac lorem. Ut suscipit ante eget magna. Morbi luctus aliquet odio.

1.1 Subsection

Aenean turpis velit, ullamcorper sed, viverra vel, consectetuer sit amet, [1] Chenipsum. Phasellus sed lectus. Vivamus fermentum odio sed odio. Donec a dui. Duis et neque quis ligula pulvinar porttitor. Nunc mattis lectus vitae diam.

Praesent quis orci. Aliquam id urna. Sed dolor erat, faucibus et, mattis eget, **commodo** nec, lorem. Etiam sit amet nisi sit amet nisi posuere bibendum. *Cum sociis natoque* penatibus et magnis dis parturient montes, nascetur ridiculus mus.

- Aliquam
- mus
- montes

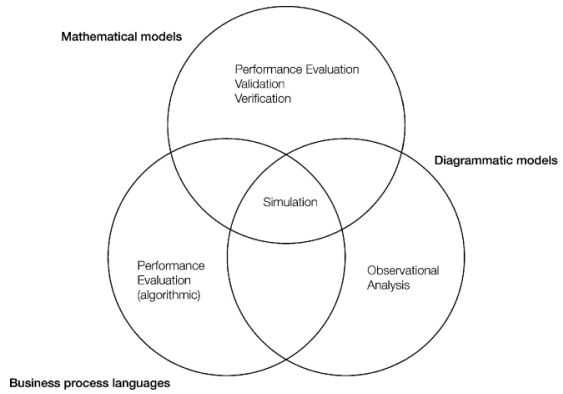

Figure 1.1: caption text

Table 1.1: Table caption text

X	Y
Item1	description
Item2	description
Item2	description

Subsubsection

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Vivamus elementum sem eget tortor. Pellentesque id orci cursus sem tempus porttitor. Aenean tincidunt, neque vitae bibendum lacinia, magna erat dapibus **Table 1.1** nunc, vel pharetra nibh erat ac lorem. Ut suscipit ante eget magna. Morbi luctus aliquet odio. Aenean turpis velit, ullamcorper sed, viverra vel,

2 Richtiges Zitieren

1. Die Seminararbeit ist eine eigenständige wissenschaftliche Arbeit und wird auch nach den Regeln einer wissenschaftlichen Arbeit erstellt (vgl. [2]), insbesondere heißt das, dass die Regeln für:

   1. Richtiges Zitieren
      - Zitierpflicht
      - Zitierregeln
      - Typen von Zitaten
      - Zitierformen
   2. Literaturangaben
   3. eine gut strukturierte Arbeit

beachtet und eingehalten werden.

3 Chapter

Duis porta orci. Integer eu arcu at enim tempus facilisis. Pellentesque dignissim orci sed est. Etiam elementum laoreet mi. Donec nunc sapien, dictum in, tristique sed, aliquam vitae, massa. Morbi magna magna, vestibulum tempor, lobortis non, convallis nec, nibh. In sed nibh. Suspendisse adipiscing dictum pede. Suspendisse non augue. Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Pellentesque lacinia, velit sed commodo convallis, diam dolor consequat ligula, a scelerisque quam neque et purus. Praesent vel augue. Sed lectus leo, dignissim eget, vulputate eu, auctor ut, nulla. Vivamus a quam. Nulla tellus. Pellentesque tempor pulvinar nunc.

4 Testing functional and nonfunctional requirements in User Requirements Notation

4.1 Introduction

Software testing is a key-activity for software quality management to ensure that the developed software behaves as expected. The anticipated behavior and requirements can be defined in the User Requirements Notation.

The User Requirements Notation is a notation for modeling, analyzing, and controlling the correctness and completeness of functional and non-functional requirements. It is divided into two notations: the Use Case Map for the functional and the Goal-oriented Requirements Language for the non-functional requirements. User Requirements Notation is a good starting point for several reasons: it is scenario-based, requirements oriented, independent of the implementation, model functional and non-functional requirements, and is easy to add in the development process. Independence from the implementation is essential; even though code-based testing is an excellent test method, the stakeholder's requirements will not be tested.

A serious risk of manual test generation is a high probability for false test cases. This is due to the test cases' repetition, whereas only the input and output data change. Therefore, the test logic is the same, and the developer tends to copy and paste the test cases, which can lead to copying the code and the test logic mistakes. In this case, writing manual tests is time-wasting for the developer. To avoid these risks and use the developer's time wisely, it is essential to automate the test case generation process.

This chapter focuses on the article *"Scenario-Based Validation Beyond the User Requirements Notation" by Arnold et al.*[3] and the article *"Transforming Workflow Models into Automated End-to-End Acceptance Test Cases" by Boucher and Mussbacher*[4]. The primary was provided by the supervisors, whereas the second was selected after the literature research. Further information about the literature research is presented in the second section. In the third section, the first paper's approach is described and applied, which will also be the case in the fourth section regarding the second paper. In the penultimate section, the two approaches will be compared based on a synthesis matrix, and the chapter will be completed with a conclusion.

Please visit the glossary to get familiar with: functional requirements (FR), non-functional requirements (NFR), User Requirements Notation (URN), Use Case Map (UCM), Goal-oriented Requirements Language (GRL), acceptance test, test cases, stakeholders, scenario, implementation under test (IUT), JUnit testing Framework.

4.2 Literature Search

Research planning

Research question: The research question is "Which approaches for the systematic generation of tests for functional and non-functional requirements from the User Requirements Notation exist?"

Snowballing: For the snowballing research, the given paper *"Scenario-Based Validation Beyond the User Requirements Notation" by Arnold et. al* will be used. The paper is founded on IEEE Xplore.[1] On IEEE Xplore, one can see that the article referenced 29 articles and has been referenced by four.

Search term based search: The search was done for the terms "User Requirements Notation" and "test" in order to answer the research question. If the papers do not include in *"All Metadata"* those two terms, it is not useful to answer the research question.

Research Sources: IEEE Xplore[2], ACM[3]. These two are the primary scientific associations for computer science and include almost all essential and verified scientific papers.

Relevance criteria: The paper should deal with an approach for the systematic generation of tests for functional and non-functional requirements from the User Requirements Notation. In detail:

- User Requirements Notation: for our research question, the requirements must be written down in the User Requirements Notation
- functional and non-functional requirements: both the functional and non-functional requirements should be tested
- systematic generation of test: the goal is that the tests are systematically generated
- extra: the article should not be by the same authors

Research results (table 4.1)

Backward snowballing: *Arnold et al.* have 29 references. 13 of the 29 references can be ignored because they are only references for definitions and links to websites for more information. Thus, 16 articles are helpful to address the research question, which is why all 16 papers are analyzed and checked if the reference criteria are met. Unfortunately, non of them meets all criteria; either the articles deal with User Requirements Notation and functional and non-functional requirements or the systematic generation of tests.

Forward snowballing: The article from *Arnold et al.* was published in the year 2010, according to IEEE Xplore. Since then, only four articles referenced *"Scenario-Based Validation Beyond the User Requirements Notation"*. One of the four is a summary of 35 articles - another is written by one of the authors of the given paper, and the third does not meet any criteria. One article left *"Transforming Workflow Models into Automated End-to-End Acceptance Test Cases" from Boucher and Mussbacher*. The last article deals with the systematic generation

[1] https://ieeexplore.ieee.org/document/5475050
[2] https://ieeexplore.ieee.org/Xplore/home.jsp
[3] https://dl.acm.org/

of tests for functional requirements from the User Requirements Notation. However, the generation of tests for non-functional requirements is not addressed by *Boucher and Mussbacher*, an article that meets all the decision criteria was not found, which is why the last paper was chosen for future evaluation.

Search term based search: On IEEE Xplore, on searches with *"(("All Metadata":"user requirements notation") AND "All Metadata":test)"* and gets five results. One result is the given article *"Scenario-Based Validation Beyond the User Requirements Notation"* and another the article *"Transforming Workflow Models into Automated End-to-End Acceptance Test Cases"* from the forward snowballing, so we get three new papers. Unfortunately, none of the three papers meets all criteria. On ACM, one searches with *"[Abstract: "user requirements notation"] AND [All: test]"* and gets seven results. One of them was *"Transforming Workflow Models into Automated End-to-End Acceptance Test Cases"*, and of the remaining six papers, none meets all criteria.

Table 4.1: Literature Research Documentation.

Source	**Date**	**Restrictions**	**Term**	**Results**	**Relevant**	**Used**
IEEE*	22.11.2020	none	backward snowballing	16(29)	0	none
IEEE*	22.11.2020	none	forward snowballing	4	1	∇
IEEE*	22.11.2020	none	(("All Metadata":"user requirements notation") AND "All Metadata":test)	3(5)	1	none
ACM†	22.11.2020	urn: abstract	[Abstract: "user requirements notation"] AND [All: test]	6(7)	1	none

**: https://ieeexplore.ieee.org/Xplore/home.jsp †: https://dl.acm.org/ ∇: [4]*

4.3 Scenario-Based Validation Beyond the User Requirements Notation[3]

4.3.1 Description

Dave Arnold and *Jean Pierre Corriveau* of Carleton University and *Wei Shi* of University of Ontario Institute of Technology wrote the paper *Scenario-Based Validation Beyond the User Requirements Notation*, which was published in 2010 by IEEE in the 21st Australian Software Engineering Conference.

The authors want to automate the generation, the execution, and the evaluation of test cases for functional and non-functional requirements from the User Requirements Notation (see figure 4.1). The automatic test generation should be independent of the implementation under test (IUT), but the test case executability on the IUT must be secured. Additionally, *Arnold et al.* require traceability between the test cases and the IUT in order to find out where the error occurs. To achieve these goals, they define the "traceable requirements model" (TRM), which supports traceability by connecting its elements with the IUT elements and also connects the stakeholders with the developers. Moreover, the TRM can handle functional and non-functional requirements, support metric evaluators to evaluate the non-functional requirements, and statistic checks for the statistical analysis. TRM is textual and must be written in its requirements specification language with the name "Another Contract Language". The transformation from the User Requirements Notation, in detail the Use Case Map and Goal-oriented requirements Notation, to the TRM is not automated. This process requires a person who is familiar with the Another Contract Language's semantic, the test fall generation process, and the binding process between the TRM and the IUT. The authors note that this process is straightforward because of the semantic similarity between the URN and the TRM. After the TRM has been created, the developer can start the "Validation Framework" (VF), which needs the TRM and the IUT as input. The VF's first step is to call the "automated binding engine", which automates the binding process between the TRM and the IUT. This process must be done to enable the test cases' executability on the IUT and traceability between the test cases and the IUT. The VF's next step is to start the IUT and run the statistic checks and the metric evaluator. After that, the VF can test the IUT with the already created test cases and check if it behaves as expected. The VF stores information about the input of the test case, the behavior of the IUT, the difference to the expected behavior, if necessary, and the points where the IUT begins to behave unexpectedly as well as the results. All of this information will be written in the "contract evaluation report".

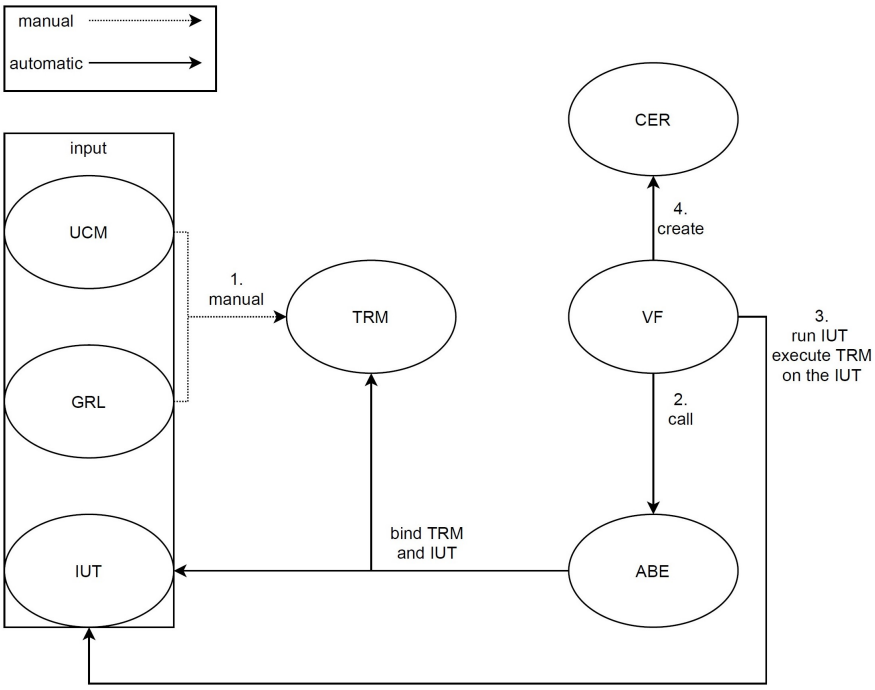

Figure 4.1: Flow of approach 1

4.3.2 Application

Input: User Requirements Notation.

The functional requirements in the Use Case Maps Notation and non-functional requirements in the Goal-oriented Requirements Language are in view in figure 4.2 and figure 4.3.

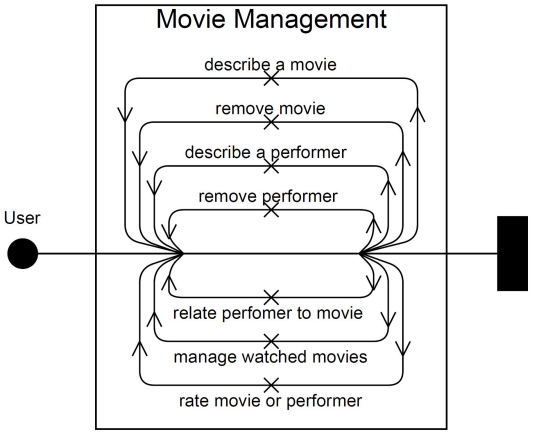

Figure 4.2: Functional Requirements in Use Case Maps Notation

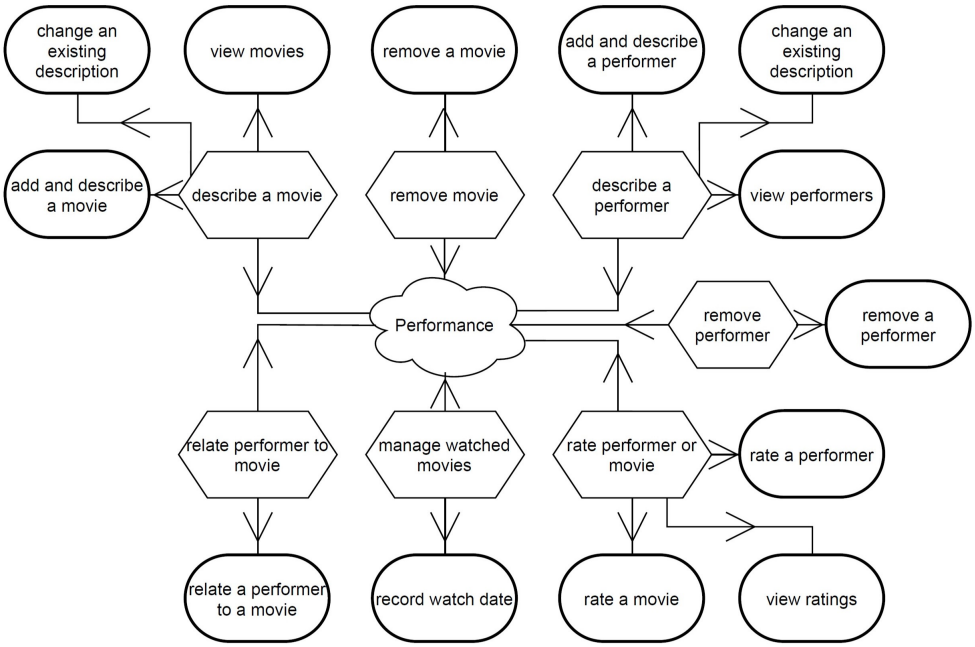

Figure 4.3: Non-functional Requirements in Goal-oriented Requirements Language

First Step: Conversion URN to TRM

For simplicity, the example will be limited to the system functions "Add Movie" and "Remove Movie". The non-functional requirement is "performance", and the question is how fast movies can be added and removed. The code of the TRM is in view in Listing 4.1

```
Contract MovieManagment {
    Scalar Integer numberOfMovies;
    Scalar Timer add_timer;
    Scalar Timer remove_timer;
    Observability Boolean HasMovie(Movie item);
    Responsibility AddMovie(Movie item) {
        Pre(HasMovie(item) == false);
        add_timer.Start(item);
        Execute();
        add_timer.Stop(item);
        numberOfMovies = numberOfMovies + 1;
        Post(HasMovie(item) == true); }
    Responsibility RemoveMovie(Movie item) {
        Pre(HasMovie(item) == true);
        remove_timer.Start(item);
        Execute();
        remove_timer.Stop(item);
        numberOfMovies = numberOfMovies - 1;
        Post(HasMovie(item) == false); }
    Scenario AddRemoveMovie {
        once Scalar Movie x;
        Trigger(AddMovie(x)), Terminate(RemoveMovie(x)); }
    Scenario Lifetime {
        Trigger(new()), (Add(dontcare)|Remove(dontcare))*,
        Terminate(finalize()); }
    Metric List Integer TimesToAddMovie() {
        add_timer.Values();  }
    Metric List Integer TimesToRemoveMovie() {
        remove_timer.Values(); }
```

```
30      Reports {
31          ReportAll("The average add time is: {0}",
32          AvgMetric(TimesToAddMovie()));
33          ReportAll("The average remove Time is: {0}",
34          AvgMetric(TimesToRemoveMovie()));}
35      Exports {
36          Type Movie { not context; } }
37  }
```

Listing 4.1: TRM

Second Step: Run VF

The VF first calls the ABE for the binding process between corresponding elements of the TRM and the IUT. Result in table 4.3

Table 4.3: Binding table between TRM and IUT

TRM element name	TRM type	IUT bindpoint	IUT type
MovieManagement	Contract	MovieManager	class
Movie	Exported Type	Movie MovieManger::Movie	class
Boolean HasMovie(Movie item)	Observability	bool MovieManager::Movie.HasItem (MovieManager::Movie)	Method
Void AddMovie(Movie item)	Responsibility	void MovieManager::Movie.createMovie (MovieManager::Movie)	Method
Boolean RemoveMovie(Movie item)	Responsibility	void MovieManager::Movie.removeMovie (MovieManager::Movie)	Method

After that, the VF starts the IUT, runs the TRM on the IUT, saves all information, and creates the CER.

4.4 Transforming Workflow Models into Automated End-to-End Acceptance Test Cases[4]

4.4.1 Description

Mathieu Boucher and *Gunter Mussbacher* of McGill University wrote the paper *Transforming Workflow Models into Automated End-to-End Acceptance Test Cases*, which was published in 2017 by IEEE/ACM in the 9th International Workshop on Modelling in Software Engineering (MiSE).

The authors in this paper want to generate acceptance tests from the User Requirements Notation. Since *Boucher* and *Mussbacher* focus on the functional requirements, they only need the Use Case Maps. These test cases will be executed with the "JUnit Testing Framework", which will be discussed later. In order to enable this, the Use Case Maps must be extended. The input data, the expected output, a description of the behavior, and the test logic with postconditions are needed. In greater detail, the scenario groups, scenario definition, start- and endpoints, and responsibilities have to be included. Scenario groups contain the name of the data/class, global settings and information for the starting and ending process, and global variables for the tests. A scenario group includes several scenario definitions, which correspond to a use case. The scenario definitions' inputs consist of the data type (e.g., boolean, integer, string, etc.) and the values. Start points define the system state before the scenario starts. Responsibilities describe an action in the scenario and are connected with the system code, whereas endpoints define the system state after the scenario ends.

The test case's extent depends on the number of responsibilities that the traversal mechanism meets during the traversal. If the mechanisms only meet one responsibility, the test cases are like Unit Tests. If the mechanisms meet many responsibilities, the test cases are similar to acceptance test cases. After the Use Case Maps diagram is extended, the traversal mechanism can analyze the diagram, which means it is traveling through the Use Case Map, from the start point to an endpoint by visiting the responsibilities, which creates the test cases automatically. For each scenario definition, the mechanism will generate a test case scaffold, which is used with the input combinations to create specific acceptance test cases. Thus, one has one test case for each input combination.

The input combinations are determined with boundary value analysis and Myer's test selection heuristics. In further detail, for instance the input data is from type integer and has the interval [-10,20], which results in eight test cases, two invalid {-11, 21} and six valid {-10,-9, 0, 5, 19, 20}. In *"TABLE I. SUPPORTED TYPES FOR INPUT VARIABLES"* on page four, one can find information about creating the test cases from the input values for all supported data types. If there is more than one input data for a scenario definition, the values' selection is based on Myer's test selection heuristics. They create one test case for each invalid value, and the other input values must be valid. Therefore, one can focus on each invalid value. Each input's valid values are combined for the rest, so all the valid values are tested at least once. The test cases end with comparing the output and the system's behavior with the expected output and behavior. If all test cases are created, the "JUnit Testing Framework" can execute the test cases on the software and show which are successful and which not.

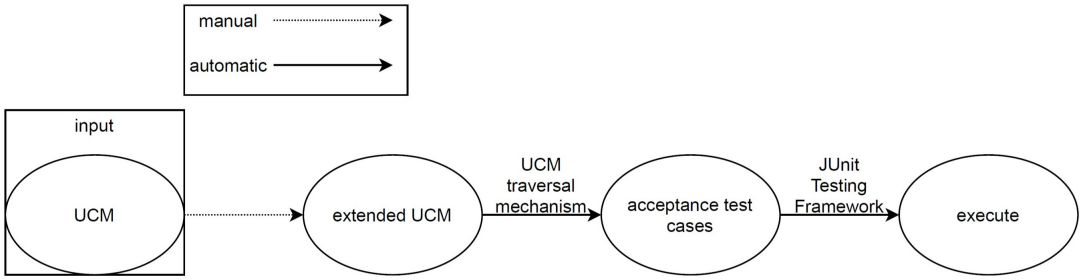

Figure 4.4: Flow of approach 2

4.4.2 Application

Input: Use Case Map

For simplicity, the example will be limited to the system function "add and describe a movie". The Functional Requirements in the Use Case Maps Notation is in view in figure 4.5.

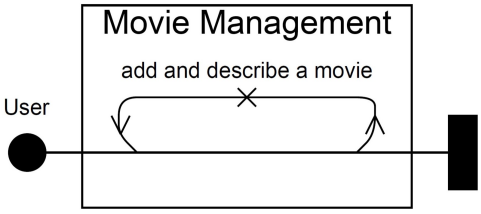

Figure 4.5: Functional Requirements in Use Case Maps Notation

First Step: Extend the UCM

The extended UCM is on view in figure 4.6.

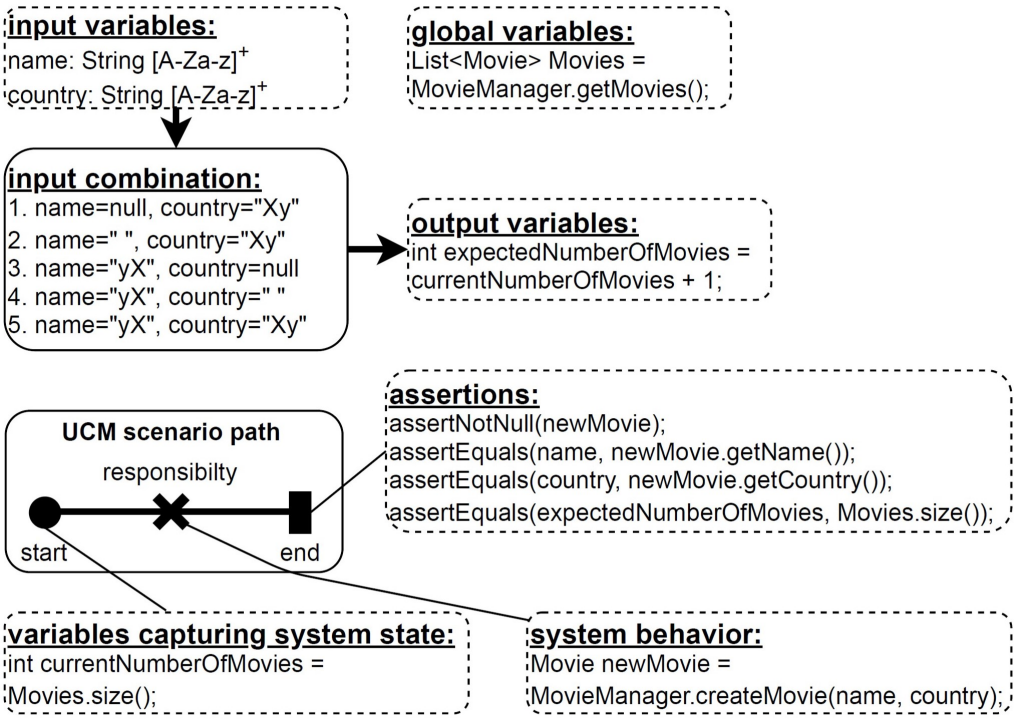

Figure 4.6: Extended Use Case Map

Second Step: Run traversal mechanism

The acceptance test cases are now automatically generated from this extended UCM model with the traversal mechanism. Result in Listing 4.2

```
1  public class MovieManager {
2      //global variables
3      List<Movie> Movies = MovieManager.getMovies();
4      @BeforClass //method TODO
5      @Befor //method TODO
6      @After //method TODO
7      @AfterClass //method TODO
8      @Test
9      public void nameNullTest() {
10         String name = null;
11         String country = "Xy";
12         int currentNumberOfMovies = Movies.size();
13         try{
14             Movie newMovie = MovieManager.createMovie(name, country);
15             fail();
16         } catch (Exception exception) {} }
17     @Test
18     public void nameEmptyTest() {
19         String name = "";
20         String country = "Xy";
21         int currentNumberOfMovies = Movies.size();
```

```
22          try{
23              Movie newMovie = MovieManager.createMovie(name, country);
24              fail();
25          } catch (Exception exception) {} }
26      @Test
27      public void countryNullTest() {
28          String name = "yX";
29          String country = null;
30          int currentNumberOfMovies = Movies.size();
31          try{
32              Movie newMovie = MovieManager.createMovie(name, country);
33              fail();
34          } catch (Exception exception) {} }
35      @Test
36      public void countryEmptyTest() {
37          String name = "yX";
38          String country = "";
39          int currentNumberOfMovies = Movies.size();
40          try{
41              Movie newMovie = MovieManager.createMovie(name, country);
42              fail();
43          } catch (Exception exception) {} }
44      @Test
45      public void createMovieValidEquivalenceClassesTest() {
46          String name = "yX";
47          String country = "Xy";
48          int currentNumberOfMovies = Movies.size();
49          try{
50              Movie newMovie = MovieManager.createMovie(name, country);
51          } catch (Exception exception) {
52              fail(); }
53          int expectedNumberOfMovies = currentNumberOfMovies + 1;
54          assertNotNull(newMovie);
55          assertEquals(name, newMovie.getName());
56          assertEquals(country, newMovie.getCountry());
57          assertEquals(expectedNumberOfMovies, Movies.size()); }
58  }
```

Listing 4.2: JUnit test cases

Third Step: Run JUnit Testing Framework

Now, the JUnit Testing Framework can execute the acceptance tests.

4.5 Comparison

The approaches have been compared using a set of synthesis questions, as shown in table 4.4 and table 4.6

Both approaches start with the User Requirements Notation. Approach one deals with the functional and non-functional requirements, whereas approach two only deals with the functional requirements. Nevertheless, in both cases, one must process the User Requirements Notation manually. In the first, the User Requirements Notation has to be transferred to the Traceable Requirements Model. In the second, the Use Case Maps must be extended. In theory, the transfer to the Traceable Requirements Model is more time-consuming than the extension. However, approach one has the advantage of testing the non-functional requirements as well. From this point, everything is automated. The first approach starts with the Validation Framework, which connects the Traceable Requirements Model and the implementation under test, executes the test cases, collects information, and creates the Contract Evaluation Report. The second approach generates the acceptance test cases with the traversal mechanism and runs the test cases with JUnit. A significant difference is that after *Arnold et al.*, the information was evaluated and written in a report, whereas after *Boucher* and *Mussbacher*, the result only includes which acceptance test cases are successful and which not. Therefore, the first one provides more information. Both help the stakeholder and developer in the testing process and creates a time advantage compared to the manual test creation. If one wants to test automatically functional and non-functional requirements, one has to use method one. However, both ways can be used if only the functional requirements are in question.

Table 4.4: Synthesis Matrix part 1/2.

Question	Name	Approach 1: **Scenario-Based Validation Beyond the User Requirements Notation**	Approach 2: **Transforming Workflow Models into Automated End-to-End Acceptance Test Cases**
1 **Description**	a) artefacts and relationship between artefacts	- "Implementation under test" (IUT): Part of the software to be tested. - "User Requirements Notation" (URN): Notation for functional requirements (FRs) and non-functional requirements (NFRs). The FRs are written in "Use Case Maps" and the NFRs in "Goal-oriented requirements Language". - "Traceable Requirements Model" (TRM): Generate test cases for FRs and NFRs. Enable traceability while linking its elements to related elements of an IUT. It is written in its language "Another Contract Language". - URN and TRM are semantically similar. - "Automated Binding Engine" (ABE): Automate the binding process between the elements of the TRM and the related elements of an IUT - "Validation Framework" (VF): Get the IUT and the TRM. Call ABE to bind the elements of TRM and IUT. Run the IUT, test the behavior of the IUT with the test cases from TRM. Collect information about the input, the behavior, and the output. Evaluate the information with the metric evaluator and create the CER. - "Contract Evaluation Report" (CER): Report with all Information. The output of the VF.	- "Use Case Maps" (UCM): Part of the User Requirements Notation. Notation for functional requirements (FRs). - "Extended UCM": Include scenario groups, which are defined with global settings and information about setup and tear down. A scenario group consists of several scenario definitions, and every definition corresponds to a use case, which should be tested. The scenario definitions need information about the input data type, the input data, and the expected output. - "UCM traversal mechanism": Traverse the extended UCM and create the acceptance test cases. - "JUnit Testing Framework": Execute the acceptance tests on the software.
	b) preconditions/ input	- FRs and NFRs in URN - IUT	- FRs in URN
	c) steps, results, informations	- The requirements must convert manually from the URN to the TRM. - The VF must be started. VF calls the ABE for the automated binding process. VF runs the IUT, tests the behavior of the IUT with the test cases, collects and evaluates the information, and creates the CER.	- The UCM must be extended with information about scenario groups, scenario definition, start-/endpoints, and responsibility. - The UCM traversal mechanism creates the acceptance test cases by traverse the extended UCM. - "JUnit Testing Framework": Execute the acceptance tests on the software.

Table 4.6: Synthesis Matrix part 2/2.

Question	Name	Approach 1: **Scenario-Based Validation Beyond the User Requirements Notation**	Approach 2: **Transforming Workflow Models into Automated End-to-End Acceptance Test Cases**
2 **Benefits**	a) supported usage scenarios	- The testing process for the FRs and NFRs is automated, and there is traceability between the test cases and the IUT.	- The testing process for the FRs is automated.
	b) supported stakeholders	- All stakeholders who are responsible for testing the software, both internal and external stakeholders.	- All stakeholders who are responsible for testing the software, both internal and external stakeholders.
	c) corresponding SWEBOK-Knowledge Areas	- Software Requirements: 1.3 Functional and Nonfunctional Requirements, 6.3 Model Validation - Software Testing: 3.6 Model-Based Testing Techniques, 6.1 Testing Tool Support - Software Engineering Models and Methods: 1.4 Preconditions, Postconditions and Invariants, 2.2 Behavioral Modeling, 3.4 Traceability	- Software Requirements: 1.3 Functional and Nonfunctional Requirements, 6.4 Acceptance Test - Software Testing: 3.6 Model-Based Testing Techniques
3 **Tools**	a) tool support	- VF*	- "UCM traversal mechanism" from "jUCMNav tool"$^{\nabla}$ - "JUnit Testing Framework"†
	b) level of automation	- The conversion from the URN to the TRM must be done manually without tool support. It requires a person who is familiar with a) the semantic of ACL, b) the test case generation process, and c) the binding process between IUT and TRM. - The binding process, the execution of the IUT, the test process, and the report creation is all automated by the VF.	- The extending of the UCM must be done manually. - The UCM traversal mechanism automates the acceptance test generation from the extended URM. - "JUnit Testing Framework" execute the acceptance tests.
4 **Quality**	a) evaluation	- The authors did not evaluate the approach.	- They have compared the number of codes of the extending process with the number of codes of the manually generated test cases. This was made on example software.
	b) evaluation results	- The authors did not evaluate the approach.	- They need 60 LOC for the extension of the UCM and 600 LOC for the manual test case creation.

*: The VF is developed by one of the authors, Dave Arnold, for his Ph.D. Thesis. According to the paper, the VF can be found under http://vf.davearnold.ca/. Unfortunately, Dave Arnold's website is offline, and we didn't find a copy of the VF on the internet. Under the following link, one can see the user guide of the VF https://www.yumpu.com/en/document/read/39418604/user-guide-validation-framework-dave-arnold

∇: According to the paper, the "jUCMNav tool" can be found under http://jucmnav.softwareengineering.ca/jucmnav. Unfortunately, the website is unreachable, "jUCMNav tool" can be found on https://github.com/JUCMNAV.

†: https://junit.org/junit5/

4.6 Conclusion

This chapter acquaints two approaches to automatically generating test cases for functional and non-functional requirements from User Requirements Notation. There is a lot of potential behind this idea, but unfortunately, as one can see in the literature research, this topic has not been researched enough. Perhaps this is due to the fact that one can not directly generate the test cases automatically from the User Requirements Notation. Instead, one has to do an intermediate step and learn that this extra effort is worthwhile because the developers save time by the test case generation and improve the testing process, which results in better software quality.

Unfortunately, the approach by *Arnold et al.* is the only one that addresses functional and non-functional requirements. The problem here is that the Validation Framework can not be found on the web because the link[4] from the paper to the website is not available. However, with "Wayback Machine - Internet Archive"[5] a copy of the website[6] is available. Additionally, under this copy, one can not download the Validation Framework, and without it, this approach does not work.

As stated, the approach by *Boucher* and *Mussbacher* only generates test cases for functional requirements. The frameworks, jUCMNav[7] and JUnit[8], which are needed for the approach, are available on the web. The whole process can be done quickly without problems and is recommendable.

If one wants to focus on the non-functional requirements, it can be found in the chapter *"Testing non-functional requirements with risk analysis"* and *"Testing non-functional requirements with aspects"*.

[4] http://vf.davearnold.ca/
[5] http://web.archive.org/
[6] https://web.archive.org/web/20161017193123/http://vf.davearnold.ca/
[7] https://github.com/JUCMNAV
[8] https://junit.org/junit5/

5 Conclusion

Fusce vitae quam eu lacus pulvinar vulputate. Suspendisse potenti. Aliquam imperdiet ornare nibh. Cras molestie tortor non erat. Donec dapibus diam sed mauris laoreet volutpat. Sed at ante id nibh consectetuer convallis. Suspendisse diam tortor, lobortis eget, porttitor sed, molestie sed, nisl. Integer enim nisl, lacinia in, pretium eu, viverra a, odio. Quisque at quam eget risus placerat porttitor. Suspendisse convallis, elit vitae mattis pharetra, orci nisl ultrices sapien, ac interdum metus lorem iaculis diam. Nunc id nunc sit amet nisl tincidunt congue. Curabitur et sapien.

6 Bibliography

[1] Chen, K, Zhang, W., Zhao, H.: An approach to constructing feature models based on requirements clustering. In: 13th IEEE International Conference on Requirements Engineering (RE05). pp. 31-40. (2005)

[2] Institut für Geographie Lehrstuhl für Allgemeine Wirtschafts- und Sozialgeographie: An Hinweise zum wissenschaftlichen Arbeiten. http://www.geogr.uni-jena.de/fileadmin/Geoinformatik/Lehre/backup_05_2007/pdf-dokumente/Skript_WissArbeiten.pdf

[3] Arnold, D., Corriveau, J.-P., and Shi, W., Scenario-Based Validation: Beyond the User Requirements Notation, 21st Australian Software Engineering Conf. (ASWEC 2010), IEEE CS, pp. 75–84, 2010. DOI:10.1109/ASWEC.2010.29.

[4] Boucher, M., Mussbacher, G.: Transforming Workflow Models into Automated End-to-End Acceptance Test Cases, Proc. - 2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering. MiSE 2017, pp. 68–74, 2017. DOI:10.1109/MiSE.2017.5.

List of Figures

List of Tables