Yagmur Arslan, Max Edinger, Ahmet Efe, Julian Gröger, Felix Hausberger, Stefan Hickl, Tobias Koch, Jasmin Mangei, Andre Meyering

Better software through systematic testing

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

February 26, 2021

Abstract

Testing is an essential and time-consuming part of any software development. This work is a collaboration that highlights various approaches to how systematic tests can be created to make this process more efficient, clearer, easier or more successful. After the introduction, eight different approaches are discussed in separate chapters. The approaches are put into practice using the example of a movie management software.

The first topic dealt with is "systematic generation of acceptance tests that are executable with FitNesse". This approach tries to solve the difficulties of requirement communication between software stakeholders and software developers. For this purpose, several artifacts like fit tables and fixture classes are created and used to run test cases with FitNesse. In conclusion, the approaches can work well depending on the project size and especially the amount of experience. The next chapter discusses the need for generating test cases in early phases of development. "Transition systems" can help to derive test cases from specifications. The approaches examined here offer industry-standard solutions, but the capability of generating robustness tests for fault detection is still a weakness.

The third part deals with "Testing with a timing component". The aspect particularly considered here is the temporal component. In many real-time systems, the exact time between two instruction executions or variances in the individual execution time can lead to different results. Since this approach is not very common in the industry, there are many steps that are not yet automated. In the next chapter, approaches for systems using models are examined. First and foremost, these include "classification trees". The goal is to derive automatic test cases based on the requirements and the input parameters of a model. In conclusion, classification trees are described as well arranged and close to the requirements, but can potentially produce many test cases. Moreover, it is difficult to apply these approaches to systems without a simulated model.

The sixth segment focuses on testing with system models and examines approaches to make testing with models better. Differences between system models and test models are described. After that, the difficulties using the tools given in the literature are presented. Nevertheless, it is possible to reduce test time and errors using them. The initial problem for the next topic with "Testing functional and nonfunctional requirements in User Requirements Notation" is the risk for wrong test cases during manual test creation. User requirements notation helps to fulfill requirements during test creation. Therefore, incorrect or incomplete tests are prevented. The problem here is that this approach is largely unexplored. Making the effort to take additional steps while generating more test cases, it is still possible to achieve better results.

In "Testing Non-Functional Requirements with Risk Analysis" the focus is on risk analysis, because it is precisely here that error-prone components can be discovered. Suitable approaches are then sought. Among the findings here are the importance of automated tests and that tests for non-functional requirements should have more priority. The last chapter describes "Testing Non-Functional Requirements with Aspects" and focuses on the "aspects". Aspects describe system-wide functionalities that are ioslated from the main program logic. This is supposed to help to deal with concerns at system level. The approaches considered seem promising and should be further refined even though the availability of tools is limited.

Thus, this joint work addresses the merits and difficulties of various test case generation techniques. Common problems identified are the lack of availability of tools or research, but in suitable use cases most approaches help in improving or automating the test cases. A final conclusion on this is drawn at the end of this work.

Abstrakt (Deutsch)

Testen ist ein wichtiger und aufwendiger Bestandteil jeder Softwareentwicklung. Diese gemeinsame Arbeit stellt in 8 Kapiteln verschiedene Methoden vor, wie systematische Tests effizienter, nachvollziehbarer, einfacher oder erfolgreicher erstellt werden können. In diesen Kapiteln werden die untersuchten Ansätze auf das Beispiel eines Movie Managers angewendet.

Zunächst geht es um "Systematic Generation of acceptance tests that are executable with FitNesse". Dieser Ansatz versucht die Schwierigkeiten bei der Kommunikation unter den Beteiligten zu lösen. Dafür werden auch mehrere Artefakte wie die Fit-Tabellen erstellt und herangezogen. Die Ansätze können je nach Projektgröße und Erfahrung gut funktionieren, was meistens eher von Letzterem abhängig ist, aber gerade von letzterem leider auch stark abhängig ist. Das nächste Kapitel erörtert, dass es bereits in frühen Phasen der Entwicklung möglich sein sollte, Testfälle zu generieren. "Transition systems" können dabei helfen, Testfälle aus Spezifikationen abzuleiten. Die hier untersuchten Herangehensweisen bieten Lösungen auf Industrie-Standard, allerdings ist die Umsetzbarkeit von Robustheitstests für die Fehlerfindung noch eine Schwäche.

Das dritte Kapitel beschäftigt sich mit "Testing with a timing component". Hier ist die zeitliche Komponente wichtig, da in vielen Echtzeitsystemen die genaue Zeit zwischen zwei Befehlsausführungen zu unterschiedlichen Ergebnissen führen kann. Da dieser Ansatz in der Industrie nicht sehr weit verbreitet ist, gibt es viele Arbeitsschritte, die noch nicht automatisiert sind. Im nächsten Kapitel werden Ansätze für Systeme mit Modellen betrachtet. Dazu gehören an erster Stelle die "classification trees". Die Nutzbarkeit von Klassifikationsbäume wird untersucht. Eine Schlussfolgerung ist die Schwierigkeit, diese Ansätze auf Systeme ohne simuliertes Modell anzuwenden.

Das sechste Kapitel legt den Fokus auf das Testen mit Modellen. Unterschiede zwischen Systemmodellen und Testmodellen werden beschrieben. Die Schwierigkeiten bei der Nutzung der in der Literatur angegebenen Tools werden herausgearbeitet, Testzeit und Fehler können dennoch reduziert werden. "Testing functional and nonfunctional requirements in User Requirements Notation" beschäftigt sich mit dem Risiko falscher Testfälle bei der manuellen Testerstellung. "User requirements notation" soll dabei sicherstellen, dass Anforderungen bei der Testerstellung erfüllt werden. Falsche oder unvollständige Tests sollen somit verhindert werden. Allerdings ist dieser Ansatz weitgehend noch nicht erforscht. Geht man jedoch zusätzliche Schritte bei der Testfallgenerierung, kann man trotzdem eine bessere Qualität erhalten.

Bei "Testing Non-Functional Requirements with Risk Analysis" liegt der Schwerpunkt auf der Risikoanalyse, weil genau hier fehleranfällige Komponenten entdeckt werden. Automatisierte Tests sollen bei "non-functional requirements" eine höhere Priorität besitzen. Das letzte Kapitel beschreibt das "Testing Non-Functional Requirements with Aspects" und legt den Fokus auf die "aspects". Damit sollen systemweite Funktionalitäten oder vom Restprogramm abgrenzbare Komponenten beschrieben werden. Auch hier werden die fehlende Verfügbarkeit von Tools und passender Forschung bemängelt. Die betrachteten Ansätze wirken aber vielversprechend und sollten weiter verfeinert werden.

Diese gemeinsame Arbeit geht also auf die Vorzüge und Schwierigkeiten verschiedener Verfahren zur Testfallgenerierung ein. Häufige Probleme, die herausgearbeitet wurden, sind die mangelnde Verfügbarkeit von Werkzeugen oder wissenschaftlichen Artikeln, aber in passenden Anwendungsfällen helfen die meisten Ansätze bei einer Verbesserung oder Automatisierung der Testfälle.

Contents

1 Introduction

1.1 Systematic Software Testing

With the rise of smart gadgets and the Internet of Things, more and more parts of our daily life involve technology. And the software required to run our smartphones, computers and other gadgets becomes more complex as more data can be processed. As software becomes more complex, bugs and even small configuration mistakes can have immense consequences as recent data breaches have shown[1]. Furthermore, not testing software can have legal consequences. If basic security measures are not implemented and tested, businesses may violate the "General Data Protection Regulation"[2] (GDPR) and may have to pay fines. This is not a hypothetical risk. The "GDPR Enforcement Tracker"[3] lists instances where businesses had to pay fines due to "Insufficient technical and organisational measures to ensure information security".

Therefore, proper testing of software becomes more important than ever. But what is software testing? According to the "Guide to the Software Engineering Body of Knowledge" (short: SWEBOK), "Software testing consists of the dynamic verification that a program provides expected behaviors on a finite set of test cases, suitably selected from the usually infinite execution domain." [16] This means that it may not even be possible to test every input against the expected output, for example if the input is an infinite data stream. But also for functionality with a finite set of input data, testing every possible combination may not be feasible. Software testing can take more time than developing the program under test. Hence, software testing can be tedious, difficult and expensive. So the question arises how software can be tested in an effective and efficient way.

We need to write tests in a systematic way. This paper contains 9 chapters, each describing two similar approaches to systematic software testing.

In chapter 2 we focus on acceptance tests with FitNesse. Communicating requirements between customers and developers can be difficult. Whereas natural language can be too ambiguous, code can be too technical. By using FitNesse, human-readable Fit-Tables which include test steps are created that can be understood by customers. By writing fixture classes, these Fit-Tables can be used to automatically run tests.

But having tests may not be enough. Which requirement is covered by which test? Can we be sure that the implementation matches the specification? Traceability becomes necessary and is required to not loose overview over the tests cases and covered requirements. This is where chapter 3 comes into play with transition systems which can be used to automatically create test paths through the application by using formal specifications.

[1] "235 Million Instagram, TikTok And YouTube User Profiles Exposed In Massive Data Leak", https://www.forbes.com/sites/daveywinder/2020/08/19/massive-data-leak235-million-instagram-tiktok-and-youtube-user-profiles-exposed/, last visited on 2021-02-10

[2] see https://eur-lex.europa.eu/legal-content/EN/TXT/?uri=CELEX%3A02016R0679-20160504&qid=1614251901149, last visited on 2021-02-25

[3] see https://www.enforcementtracker.com/, last visited on 2021-02-10

After that, chapter 4 looks into testing with a timing component, since for real-time systems, timing is crucial. In real-time systems, the exact time between two instructions or variances in the individual execution time can lead to different system reactions and therefore different results.

Following this, chapter 5 looks into decreasing the number of test cases by using classification trees. Because testing all possible parameter cases becomes unmanageable very fast, these classification trees offer a great way to reduce the complexity of such parameter constructs. Furthermore, these classification trees can be used to generate test cases.

chapter 6 *is missing*

In chapter 7 we then look into testing with systems models which are compared against test models. It is looked into model based testing and how traceability can be used to increase the probability of finding errors and improve test quality.

Chapter 8 explains why writing manual tests for functional and non-functional requirements can be not only tedious but error prone due to Copy & Paste of errors in test logic. The section describes how test cases can be automatically created by using the user requirements notation that is used for modeling, analyzing, and controlling the correctness and completeness of functional and non-functional requirements.

Another aspect of testing non-functional requirements can be by using risk analysis. This is where chapter 9 steps in and briefly shows how non-functional requirements can be tested and how risk analysis can be used to prioritize test cases. The section also shows how architectural non-functional requirements such as code conventions can be tested.

Chapter 10 expands on this topic by introducing aspects and aspect oriented programming to test and verify non-functional requirements such as software memory limits and memory leaks.

Finally, chapter 11 concludes this paper and summarizes each article.

1.2 Common Fundamentals

All chapters build upon a common set of fundamental definitions regarding software testing and requirements. They are listed and mapped to the individual topics in the glossary section of this report. It is highly recommended to refer to the glossary before reading a chapter or when ambiguities arise while reading a chapter. However, since it is quite extensive and also contains more topic-specific definitions, the most important terms are defined hereafter.

Like a part of the entries in the glossary section, we use the SWEBOK in its third version as a basis. Although this guide is already quite old (at least the original version from 2004), hence not fully compliant with current research results, its 15 knowledge areas and basic definitions of a body of knowledge are still useful for classifying the approaches presented in this report and establish a common set of technical terms.

Let us begin with requirements. The corresponding SWEBOK knowledge area is “Software Requirements”. Relevant sub chapters are “Software Requirements Fundamentals” and “Requirements Validation”. The guide defines requirements as “a property that must be exhibited by something in order to solve some problem in the real world. [...] An essential property of all software requirements is that they be verifiable as an individual feature as a functional

requirement or at the system level as a non-functional requirement. It may be difficult or costly to verify certain software requirements." [16] This definition already points out the difficulty of verifying requirements that necessitates the use of systematic testing techniques, as explained in Section 1.1. Moreover, it distinguishes between functional requirements, representing a feature the software is to provide, and non-functional requirements, specifying the extend of quality. Requirements need to be formulated clearly, unambiguously and quantitatively in order to implement and verify them correctly [16].

For software testing, there is no uniform definition. Therefore, the guide refers to multiple definitions from cited references. In essence, software testing is to assure that specified requirements are met by the implementation or, from a different perspective, find errors indicating that a requirement has not been met. This testing process is performed at different levels, as the requirements definition already touched upon. The SWEBOK guide distinguishes between three test levels: unit testing, verifying isolated functionalities (mostly functional requirements), integration testing, verifying the correct interaction of components and system testing, verifying the behavior of the entire system (mostly non-functional requirements). Correspondingly, these levels are distinguished by the object of the test (single module, multiple modules, entire system), called the target of the test, and the purpose, called objective of the test [16].

The guide presents a wide array of testing techniques. For this report, it is important to take notice of the definition of model-based testing: "A model in this context is an abstract (formal) representation of the software under test or of its software requirements [...]. Model-based testing is used to validate requirements, check their consistency, and generate test cases focused on the behavioral aspects of the software."[16] Some of the approaches presented in this report are model based, at least partially. However, it is not always clear what the actual model is and some authors use the term incorrectly.

1.3 Outline

Following this introduction in chapter 1, nine individual reports each present two different but related approaches for systematic testing in chapters 2 - 10. The reports introduce their superordinate topic in sections X.1, outline the results and execution of a literature search based on a given article in sections X.2 and describe the given and selected approach in sections X.3.1 and X.4.1 as well as illustrating them using a common set of given requirements in sections X.3.2 and X.4.2. Finally, the approaches are compared using a common set of synthesis questions in sections X.5 and evaluated in sections X.6. Chapter 11 concludes the report. The glossary and bibliography can be found in chapters 12 and 13.

In the following, the given requirements (Figure 1.1 on page 12) and synthesis questions, used for each individual report, are depicted.

Synthesis questions:

1. Description of the approach (What does the approach do?)

    a) Which artifacts and relations between artifacts are used in this approach? Which artifacts are created in the course of the approach? How are the artifacts characterized?

    b) What is required and/or input for the application of the approach?

    c) What steps does the approach consist of? Which information is used in which step and how? What are the results of the individual steps?

2. Benefits of the approach (Whom does the approach help and how?)

    a) Which usage scenarios are supported by the approach?

    b) Which stakeholders are supported by the usage scenarios?

    c) Which knowledge areas from SWEBOK can be assigned to the usage scenarios?

3. Tool support for the approach (What tool support is available?)

    a) What tool support is provided for the approach?

    b) Which steps of the approach are automated by a tool? Which steps are supported by a tool, but still have to be executed manually? Which steps are not supported by a tool?

4. Quality of the approach (How well does the approach work?)

    a) How was the approach evaluated?

    b) What are the (main) results of the evaluation?

User Task		Movie Management	
Purpose (Goal)		Users manage movies and corresponding performer data of a movie collection.	
Frequency		Often and at any time (depending on the user's needs)	
Actors		User who wants to manage movies	
Sub-tasks:			**Example of solution:**
1	**Describe a movie** Add and describe a movie with typical data like its title or alternative titles, release date, production country, release date, runtime, location, IMDB ID, and performers. Or change an existing description. Or view movies (possibly sorted).		Provide default values wherever possible, implemented in *create movie*, *change detail movie data*, *link existing performer*, *unlink performer*, *show movie*, *list movies*, *search*, *sort movies*, *show performer details* and *show movie in IMDB*
1ap	**Remove movie** **Problems:** Removing all movies a certain performer participates in might result in performers associated with no movies.		Ensure the consistency of performers and movies, implemented in function *remove movie*
2	**Describe a performer** Add and describe a performer featuring in movies with typical data like first and last or alternative names, biography, country, IMDB ID and date of birth. Or change an existing description. Or view performers (possibly sorted).		Provide default values wherever possible, implemented in *create linked performer*, *change detail performer data*, *link existing movie*, *unlink movie*, *list performers*, *sort performers*, *search*, *show performer*, *show movie details* and *show performer in IMDB*
2ap	**Relate performer to movie** **Problems:** Creating a performer without relating her/him to a movie lead to performers associated with no movies. Thus, a performer needs to be related to a movie.		Ensure that performers are linked to movies on creation, implemented in function *create linked performer*
2b	**Remove performer** Removes a performer.		Implemented in function *remove performer*
3	**Manage watched Movies** Record date when the movie was last watched.		Implemented in function *watch movie*
4	**Rate Movie or Performer** Rate or view ratings (sorted).		Provide a fixed rating value list. Implemented in functions *rate movie*, *rate performer*, *calculate overall rating of movie* and *sort movie*

Figure 1.1: Requirements in User-Task notation for the MovieManager software, an application for managing movie collections.

2 Systematic Generation of acceptance tests that are executable with FitNesse

2.1 Introduction

This chapter focuses on the creation of acceptance tests that are automatically executable using the tool *FitNesse*. In this first section of the chapter the reason for using this approach is discussed. Furthermore, the general features of *FitNesse* as well as the needed artifacts like Fit-tables are explained. An article that focuses on the topic of creating acceptance tests (see glossary for the term *acceptance test*) that are executable with *FitNesse* was provided by the supervisors of the seminar. Section 2.2 documents the literature search used to find a different approach. The two articles are then described in section 2.3 and in section 2.4. Moreover, for a better understanding of the presented approaches these chapters include the execution of them on the *MovieManager*. The following section 2.5 includes the comparison of the two approaches using a synthesis matrix. Section 2.6 provides a Conclusion including the most important insights of the process and an assessment in which situations the approaches might be suitable. For a definition regarding the terms acceptance test, acceptance test driven development (ATDD), stakeholders, test cases, test scenario, UML and use case please see the glossary.

During the software engineering process communication between the developers and the customers is a crucial factor for the success of the product. A problem for the communication is the different use of documents by the two main stakeholders: Customers describe their requirements in natural language whereas the developers create code. Natural language can often be interpreted in different ways, which can lead to unwanted results. And whereas code is more precise, it is often too technical for the customer. Therefore, artifacts are needed that are more precise than natural language and can easily be transformed into code.

One such artifact is a *Fit-table*. These tables store easily readable information about acceptance test cases and can be fully automatically executed using the testing tool *FitNesse* [26]. Creating *Fit-tables* before the development of the software can help the developers to understand the requirements of the customer by implementing the necessary functionality to pass the acceptance tests. Thus the customers receive a software that satisfies all their mentioned requirements. *FitNesse* supports the creation and maintenance of *Fit-tables* as well as the automated execution of the tests represented by the tables. To make this possible *Fixture-Classes* are needed. These classes connect the input values from the *Fit-tables* to the *System-under-test* and are executed by *FitNesse*. An overview of the data exchange during the process is shown in Figure 2.1 on the next page. The specific steps are explained in the following:

After the user chooses to execute a set of Fit-tables in *FitNesse*, *FitNesse* executes the Fixture-Classes that belong to the selected tables. These tables can contain two types of values: Input values and expected output values. The *Fixture-Class* creates an instance of the *System-under-test* and then transfers the input values into it. Then it extracts the resulting output values from the *System-under-test* and returns them to *FitNesse*. These extracted output values are

then automatically compared by *FitNesse* to the expected output values from the Fit-tables. If they are the same, the test was successful and the entry of the table receives the colour *Green*. Otherwise, the affected part of the test failed and the entry receives the colour *Red*.

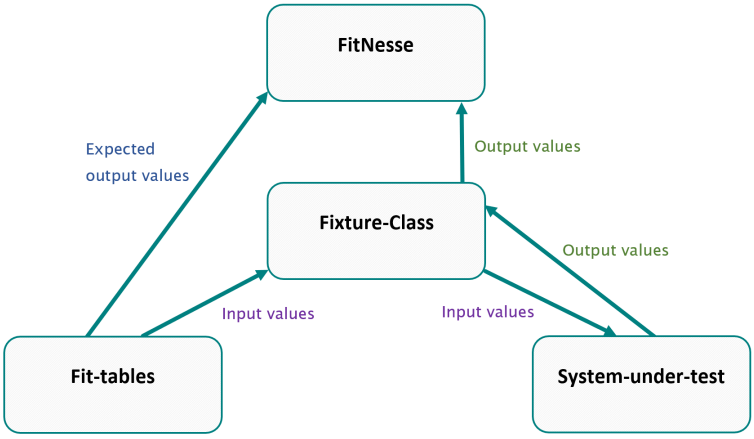

Figure 2.1: Overview of the data exchange in the execution of *Fit-tables* with *FitNesse*. The Fit-tables can be created and maintained in FitNesse.

2.2 Literature search

The literature search consisted of a search-term-based search as well as forward- and backward-snowballing. As start article the work of El-Attar and Smith [24] was given by the supervisor of this seminar. This article presents an approach to create acceptance tests that can be automatically executed using *FitNesse*. To find more and possibly different approaches the search question was chosen to be:

Which approaches to systematically generate acceptance tests that are executable with FitNesse exist in the literature?

ACM Digital Library [3], *IEEE Xplore* [34], *Springer Link* [57] and *Science Direct* [52] were used as search platforms as they are the most common platforms in the field of computer science. The relevance criteria were chosen as follows:

- *Criterion 1:* The article describes an approach to systematically generate acceptance tests that are executable with *FitNesse or* gives an overview on the use of *FitNesse* in software engineering.

  This criterion was chosen to find approaches that are specific to the subject of this chapter. Articles that give an overview over the use of *FitNesse* were also accepted because of their potential to classify the found approaches.

- *Criterion 2:* The article was not published before the year 2009 which is the year that the article by El-Attar and Smith was published.

  This criterion was chosen to get a more recent approach than the start article which was by the creation of this chapter already more than 10 years old.

Table 2.1 provides an overview for the search-term-based literature search. As search terms the terms "acceptance test" and "FitNesse" were chosen. These search terms turned out to be specific enough to fit only a manageable amount of articles. The search resulted in eight relevant articles of which six presented an approach and two gave an overview over the use of *FitNesse*. Both snowballing searches from the start articles did not result in any more relevant articles that were not already found by the search-term-based search. The backward-snowballing did not result in any relevant articles due to their publishing date and hence not passing Criterion 2. One relevant article was found during the forward-snowballing that was already found by the search-term-based search on the platform Springer Link.

Table 2.1: Overview of the search-term-based literature search.

Search platform	Search date	Search restrictions	Search terms	# results	# relevant results	# relevant new results	Used articles
ACM	18.11.20	Publishing year: 2009-2020	„acceptance test" AND fitnesse	10	4	4	**Longo et al., 2016**
IEEE Xplore	18.11.20	Publishing year: 2009-2020	„acceptance test" AND fitnesse	2	1	1	
Springer Link	19.11.20	Publishing year: 2009-2020, no preview-only content	„acceptance test" AND fitnesse	11	2	1	
Science Direct	19.11.20	Publishing year: 2009-2020	„acceptance test" AND fitnesse	9	2	2	

The articles that gave an overview over the use of FitNesse were not specific about any approaches and only provided general information. Therefore, none of these articles was used. As a second approach to compare to the start article, the work by Longo et al. [42] was chosen. This article also describes an approach to create acceptance tests that can be automatically executed by *FitNesse*. The presented approach differs from the approach of the article by El-Attar and Smith in its use of artifacts. Also it was created by different authors and was published in 2016, so it is a much more recent approach.

2.3 Approach 1: Developing comprehensive acceptance tests from Use Cases and robustness diagrams

2.3.1 Description

El-Attar and Smith [24] introduce an approach to create acceptance tests that can be automatically executed using *FitNesse*. Their approach is targeted at larger software projects that use a model-based approach such as the use of UML models (see glossary for the term *UML*). It was created such that a non-technical person (e.g. a Business Analyst) can execute it during the early phases of the development of the software. This makes it possible for the developers to follow the approach of Acceptance-Test-Driven-Development (*ATDD*, see glossary) because of the possibility of executing the acceptance tests at any time during the development process.

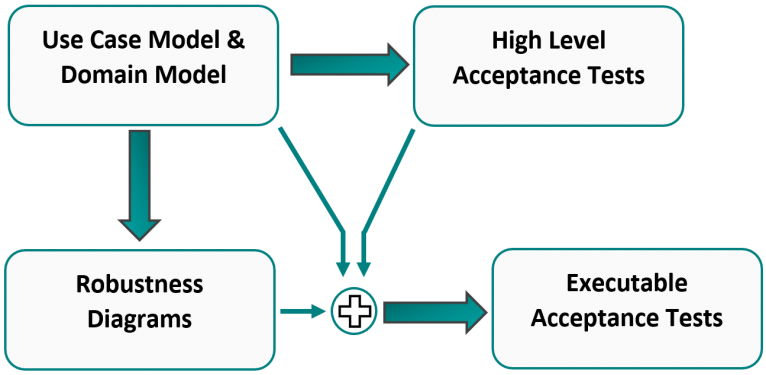

Figure 2.2: Overview of the steps in the approach of El-Attar and Smith.

ATDD helps the developers during the development process to evaluate which requirements are already implemented and which are yet to be implemented.

The approach starts with use case models and domain models as initial artifacts. During the whole process of creating the acceptance tests every step is performed completely manually. The execution of the final acceptance tests is done fully automatically by the tool *FitNesse*. To help with traceability during the approach the authors created a tool called UCAT. This tool does not provide any automation support but allows the user to create use case models and *Fit-tables*. These artifacts can be linked within UCAT which helps to determine which use cases resulted in which acceptance tests. Figure 2.2 shows the rough structure of the approach. The exact steps of the approach are described in the following:

In the first step of the approach **High-Level-Acceptance-Tests (HLATs)** are created. For this a domain model and a Use Case model with Use case descriptions is required in the approach. HLATs are more informal than an executable acceptance test which helps the analyst to be as flexible as possible in describing the acceptance tests at this early stage. Commonly Use Cases contain Use Case descriptions from which the flows of the Use Case can be extracted. HLATs describe the system's expected behaviour during all of the flows of the use cases from the Use Case model. Necessary Pre-Conditions and triggers for the flows are also extracted from the Use Case description while the inputs for the flows can be extracted from the domain model. Expected test results are also denoted in the HLATs. At this point they do not need to contain specific values and can be written in natural language. The general structure of a HLAT is shown in Table 1.2.

Table 2.2: General structure of a HLAT.

Test ID	**Description**	**Expected Result**
Name of the Use Case & the flow	Preconditions: Inputs:	Expected result in natural language

After the creation of the HLATs a robustness analysis is performed. To achieve this, for every use case a **robustness diagram** is created. These diagrams combine the use cases from the use case model with the objects from the domain model. They contain actors and entities as well as boundary- and control-objects. For each use case all involved objects and the connections between the objects are displayed. The involved objects and the communication

between them is extracted from the Use Case description. During the creation of the robustness diagrams necessary *objects* or *attributes of objects* may be identified that are not already part of the domain model. These should be added to the domain model. Also missing steps or preconditions in the Use Case description might be found. If this is the case, the Use Case descriptions should also be updated. After this step the HLATs should be adapted to fit the updated domain model and Use Case descriptions.

In the last step all the existing artifacts (possibly except the domain models) are used to create the final product of the approach: **Executable Acceptance Tests (EATs)**. These acceptance tests are in the form of specific *Fit-tables*. To achieve this, the HLATs have to be divided into smaller steps using the information about the Usage Scenario from the related use case description. This step requires human judgment and is not further described. For each of these steps the control flow in the robustness diagram gets traced. In this process the objects and attributes of each step's input, preconditions, outputs and postconditions are determined. These where before stated in natural language in the HLAT and are exchanged in the EAT with more concrete objects and attributes. The steps combined with the corresponding control flow are manually converted into *Fit-tables*. *Fit-tables* used in this approach are either *ActionFixtures*, *RowFixtures* or *ColumnFixtures* [27]. These types of *Fit-tables* can be fully automatically executed using the tool *FitNesse*. The domain models are ideally not required if the steps before where executed properly because the information from the domain models should already be part of the use case descriptions.

Due to the fact that the approach of creating the acceptance tests is done completely manually, the quality of the resulting acceptance tests depends highly on the experience and skills of the person executing the approach. Therefore, the authors state that an evaluation would be beyond the limitations of their work. However, they provide a case example by applying the approach to the software *RestoMapper*. This example is not part of this work because in the following section the execution of the approach is presented with the application *Movie Manager* that is used throughout this whole report.

2.3.2 Application

The approach starts with Use Case and Domain Models. As those are not already described in this article, they are created for this chapter. Figure 2.3 on the next page shows the Use Case Model for the Movie Manager application. It contains the use cases and shows connections between them. For example, removing a movie might result in removing a performer if one of the performers that participated in the movie has no movies anymore after the removal. Such a relation is highlighted in the Use Case Model with the keyword *extend*. The domain model of the Movie Manager application is shown in Figure 2.4 on the next page. It contains the entities Movie and Performer as well as the two views that the user can see.

To illustrate the approach the Use Case *Describe a performer* is used. In the first step of the approach the HLATs for this Use Case need to be created. The User Scenarios for this Use Case are as mentioned in the User Task table:

- Add and describe a performer
- Change the description of an existing performer
- View performers (possibly sorted)

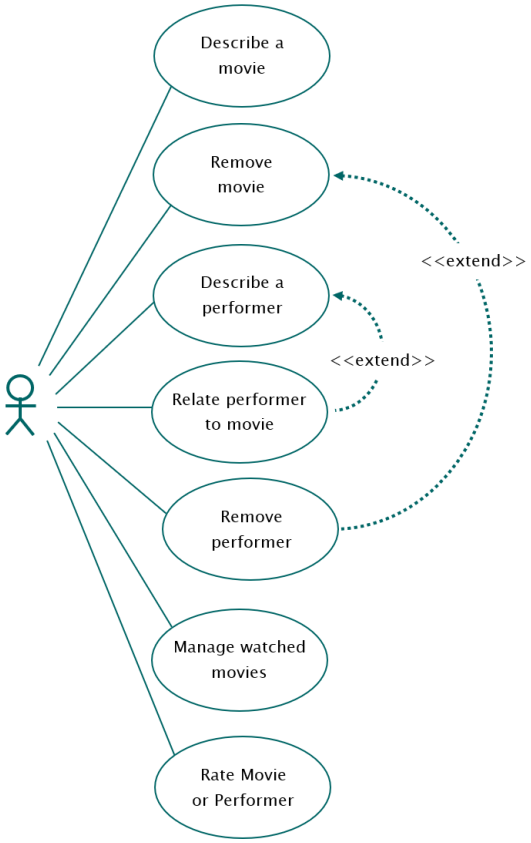

Figure 2.3: Use Case Model for the Movie Manager application.

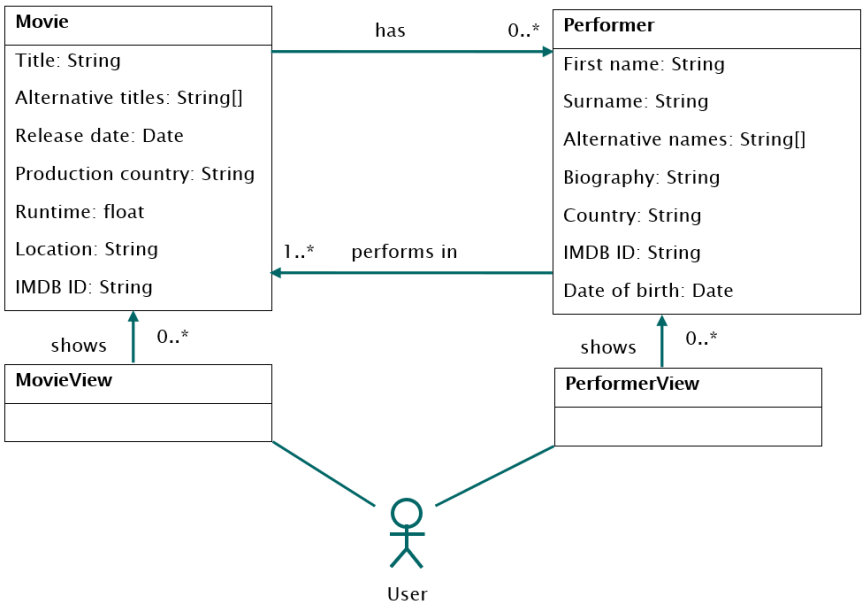

Figure 2.4: Domain Model for the Movie Manager application.

The HLATs for the Use Case *Describe a Performer* are displayed in Table 2.3. Each HLAT describes the necessary preconditions, inputs and the expected results of one User Scenario. This information is extracted from Use Case description.

Table 2.3: HLATs for the Use Case *Describe a performer* of the Movie Manager application.

Test ID	Description	Expected result
Describe-a-performer-new-performer	Precondition: a movie exists Input: values for the attributes of the performer Input: movie that the performer is linked to	The performer is created with the given attribute values. The performer can be selected. The new attribute values can be seen when the performer is selected. The performer is linked to the given movie.
Describe-a-performer-existing-performer	Precondition: a performer exists Input: performer Input: values for the attributes of the performer	The attributes of the selected performer are updated to the provided values. The new attribute values can be seen when the performer is selected.
Describe-a-performer-view-performer	Precondition: - Input: performer list Input: sorted or not	The performer list is shown. Performers are sorted if the user chooses this option.

In the next step a robustness diagram is created using the information from the Use Case Model and the Domain Model. The robustness diagram for the Use Case *Describe a movie* is shown in Figure 2.5. A robustness diagram contains the involved objects that communicate with the user. These are called boundary objects. An example for a boundary object is *PerformerView* in Figure 2.5. It also contains control-objects like *RelateToMovie* in Figure 2.5 that makes sure that a performer is related to at least one movie. The last types of objects are the *User* and the entities like *Performer* or *Movie* in Figure 2.5. The resulting robustness diagram is used to find new information for the domain diagram. For example, CheckMovieRelations needs to find out whether the performer is linked to at least one existing movie. Therefore, the domain model needs to include a list of related movies for each performer or the number of related movies. In this example the first variant (a list of related movies) is used in the domain model. So this specific information does not have to be added. Overall the robustness analysis does not bring up any new information but possibly could for other examples which is why El-Attar and Smith have included it in their approach.

As last step executable acceptance tests are created for each HLAT. These are created in the form of *Fit-tables*. For this example so called *ActionFixtures* are chosen but *RowFixtures* and *ColumnFixtures* are also possible for this approach. *ActionFixtures* contain a Test ID in the first row. Each of the following rows contains one action like entering a value or pressing a button. The ActionFixtures for the three HLATs from Table 2.3 are displayed in the Tables 2.4, 2.5 and 2.6 on the next pages. These Fit-tables are the final result of the approach.

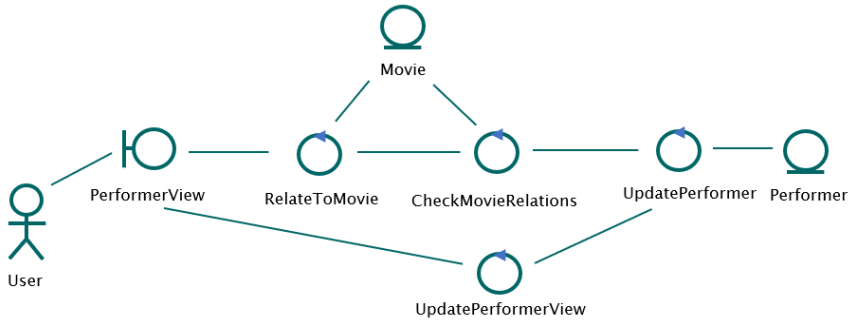

Figure 2.5: Robustness Diagram for the Use Case *Describe a performer* of the Movie Manager application.

Table 2.4: Executable Acceptance Tests for the scenario *Describe a performer, new performer* of the Movie Manager application in form of an *ActionFixture*. A placeholder in the form of ... is used for entering the other possible attributes of a performer to reduce the size of the table.

Describe-a-performer-new-performer		
Press	createNewPerformer	-
Enter	FirstName	f
Enter	Surname	s
...	...	
Enter	Biography	b
Check	Performer p with p.firstName == f and p.surname == s exists	
...	...	
Check	p.Biography == b	

Table 2.5: Executable Acceptance Tests for the scenario *Describe a performer, existing performer* of the Movie Manager application in form of an *ActionFixture*. A placeholder in the form of ... is used for entering the other possible attributes of a performer to reduce the size of the table.

Describe-a-performer-existing-performer		
Press	selectPerformer	p
Press	changeAttributes	p
Enter	firstName	f
...	...	
Enter	Biography	b
Check	p.firstName == f	
...	...	
Check	p.surname == s	

Table 2.6: Executable Acceptance Tests for the scenario *Describe a performer, view performers* of the Movie Manager application in form of an *ActionFixture*. A placeholder in the form of ... is used for entering the other possible attributes of a performer to reduce the size of the table.

Describe-a-performer-view-performers		
Press	viewPerformers	performerList
Enter	sorted	True or false
Check	performerView.performers == performerList	
Check	If sorted == True: isSorted(performerView.performers) == True	

2.4 Approach 2: A web framework for test automation, user scenarios through user interaction diagrams

2.4.1 Description

Longo et al. [42] create User Scenarios through User Interaction Diagrams (US-UIDs) which then are fully automatically converted into *Fit-tables* that represent the test data for acceptance tests. To run these acceptance tests a Fixture-Class is needed that connects the test data from the *Fit-table* with the System-under-test. The US-UIDs are created in a tool provided by the authors. They contain functional data such as the involved objects, attributes and functions and also explicit User Scenarios provided by the customer. The User Scenarios provide the test data and combined with the functional data, *Fit-tables* can be automatically created. The functional data represents the top row of the *Fit-table* and the User Scenarios the specific values. Figure 2.6 provides an overview over the steps of the approach. The only step that is executed automatically is marked in this overview. Each of the steps is explained in more detail in the following.

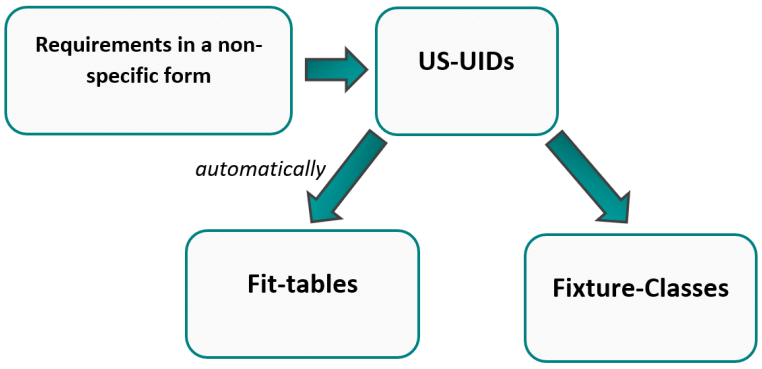

Figure 2.6: Overview of the steps in the approach of Longo et al.

In the first step the US-UIDs are created. This step is described in another work by the authors [41]. Each US-UID contains an explicit User Scenario provided by the customer. The information about the User Scenario is extended by the developer by adding functional information. For example, the User Scenario could provide a specific value for a variable. Then the functional information for this value would be the name of the variable. With this conjunction of explicit and functional values *Fit-tables* can be automatically created. The first row contains the functional information. Each row after the first row represents a User Scenario and contains the given values for the functional information (objects, variables, etc.) given in the first row.

Each row can be used as one specific test case. To execute these test cases a Fixture-Class is needed. This class has to be written by the developers. It creates an instance of the System-under-test and uses *Setter methods* to provide the input from the *Fit-table* to the System-under-test. Through *Getter methods* the resulting values of the System-under-test can be extracted to validate the success of the test. For the evaluation step the results from the Getter methods are compared to the expected values from the *Fit-table*. If they are the same, the test was successful.

To evaluate their approach Longo et al. used their tool to automatically create executable acceptance tests from existing US-UIDs. The developers of the software related to these US-UIDs already manually created test cases for the software. In the evaluation the authors compared these manually created tests to the tests created by their tool. To compare both of them the authors used the techniques *code mutation* and *lack of code*. The technique *code mutation* involved manipulating the values of an array in the software and *lack of code* was executed by removing a class from the software. By using both techniques failed tests could be found in both test sets. The second technique also resulted for both test sets in tests that were not executable. From these results the authors concluded that tests created with their approach can detect test cases that are *successful*, *failed* or *not executable*.

2.4.2 Application

To illustrate the approach of Longo et al. the use case *Describe a performer (new performer)* is used. In the first step an US-UID has to be created that displays the explicit information of a User Scenario as well as the underlying functional information. In this type of model the round boxes are states of the system. The rectangles contain the User's Input whilst the free text in the round boxes describes the system's output. Arrows are used to assign functional names to the data and to denote transitions between states. The US-UID for the example is displayed in Figure 2.7. In the beginning the performerList contains only the performers a, b and c with attributes e, f and g. After the execution of the US-UID it also contains performer d with attributes x.

In the next step the functional information from the US-UID needs to be connected to the real objects and attributes of the *System-under-test*. This is done in a Fixture-Class that is marked with *@Fixture*. This class has to be written manually. The data flow during the execution of acceptance tests with FitNesse and the role of the Fixture-Class in this process is described in section 2.1. Inputs are part of the functional data on the start of the arrows in the US-UIDs. For the first display of the US-UID PerformerView the Fixture-Class needs methods to move to the next display, to set the starting performer list and to choose *create Performer*. Results have to be extracted from the System-under-test using Getter-Methods. One such result is the updated performer list in the last state of the PerformerView. This result can be compared to the expected result given in the US-UID.

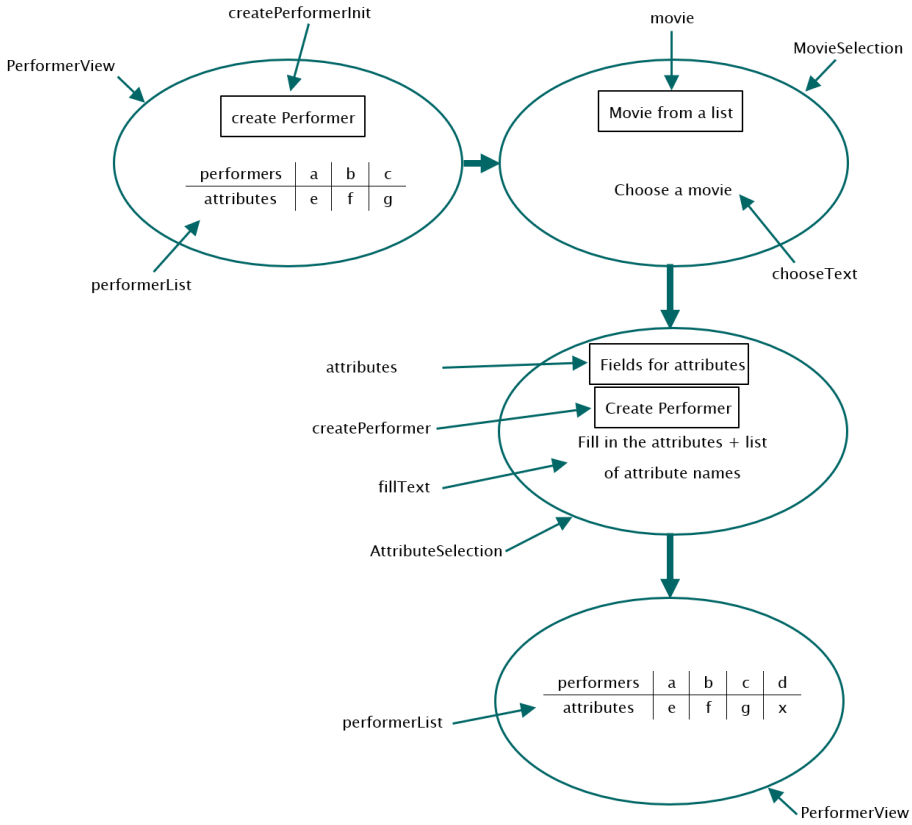

Figure 2.7: US-UID for the Use Case *Describe a performer (new performer)* of the Movie Manager application.

In the final step a *Fit-Table* is created. This step is fully automatic with the tool of Longo et al. because it is only remodeling information from the US-UID. The functional information is placed in the top row whilst the explicit data of the User Scenarios is stored in the following rows. Each row represents an User Scenario. The resulting *Fit-table* for the example is displayed in Table 2.7 on the next page.

Table 2.7: *Fit-table* for a specific User Scenario of the Use Case *Describe a performer (new performer)* of the Movie Manager application. The expected results end with a question mark.

Performer-List	Create-Performer-Init	movie	attributes	Create-Perfomer	performerList ?	d.movie ?	d.attributes ?
[a,b,c]	True	m	x	True	[a,b,c,d]	m	x

2.5 Comparison

As in the other chapters the approaches are compared in a synthesis matrix.

Table 2.8: Synthesis matrix

No.	Approach 1	Approach 2
1a)	Initial artifacts: **Use Case and Domain Models** are used to create high level acceptance tests and robustness diagrams **Robustness diagrams** combine the information from the use cases and the domain model. During the creation of the robustness diagrams for the use cases objects and attributes may be identified that are missing in the domain model. The domain model is updated with this missing information. **High level acceptance tests (HLATs)** deliver an informal description of acceptance tests. They are tables and use keywords that are chosen by their creator. For each flow of a use case from the Use Case model a HLAT is created. **Fit-tables** are a form of executable acceptance tests that can be automatically executed by the tool FitNesse. For each HLAT a Fit-table is created using the information about the control flow in the respective robustness diagram.	**User Scenarios through User Interaction Diagrams (US-UIDs)** show the interaction during a User Scenario. They include User-Input, System-Output, states of interaction and transitions between states. **Fit-tables** are a form of executable acceptance tests that can be automatically executed by the tool FitNesse. The Fit-tables in this approach need a specific Fixture-Class for each Use Case that allows the flow of information between the Fit-table and the System-under-test.
1b)	• Use Case Model • Domain Model	Requirements in a non-specific type

1c)	As initial artifacts Use Case models and Domain models are used. In the first step a HLAT is created for each flow of each use case from the Use Case model. The information about the preconditions, inputs and triggers for the HLATs gets extracted from the domain model. The second step is the creation of robustness diagrams for the use cases from the Use Case model. These diagrams also include the objects from the domain model and model the communication between those in the specific use case. If objects or attributes are found in this step that are necessary but not yet part of the domain model, they are added to the domain model. All the other models are updated to fit the domain model. In the last step a Fit-table is created for each HLAT using the control flow that can be seen in the robustness diagram. Finally the created Fit-tables can be combined with the Use Case from the Use Case model that they belong to.	In the first step the US-UIDs are created by using the known requirements. The customer delivers the User Scenario and the developer add the functional information for the data used in the scenario. The Fit-tables are created automatically from the information of the US-UIDs. This step is done by the web framework.
2a)	Business Analysts receive a systematic approach to create acceptance tests in the Fit-syntax. Customers receive a product that fits their requirements. Developers receive acceptance tests that they can use to determine which requirements of the customers they have already implemented and which they have to work on.	Customers and developers receive an approach to develop acceptance tests together that include User scenarios provided by the customer. Developers receive acceptance tests that they can use to determine which requirements of the customers they have already implemented and which they have to work on.
2b)	• Developer • Customer • Business Analyst	• Developer • Customer
2c)	Developer: Software Construction (Test-driven development) Business Analyst & Customer: Software Testing	Customer/Developer: Software Requirements, Software Testing Developer: Software Construction (Test-Driven Development)
3a)	The authors provide the tool UCAT in which Use Case Models and Fit-tables can be created and linked. The created Fit-tables can be automatically executed using FitNesse.	A web framework is provided by the authors in which US-UIDs can be created and converted to Fit-tables. These Fit-tables can be executed with FitNesse.
3b)	The tool UCAT serves as an editor to create Use Case models and Fit-tables. Those two artifacts can also be linked in UCAT. Every other step in the creation of the acceptance tests is done without tool support. No step of the approach is done automatically.	The creation of the US-UIDs is supported by the web framework which serves as an editor. Converting the US-UIDs to Fit-tables is done fully automatically by the web framework. Every other step in the creation of the acceptance tests is done without tool support.

4a)	• Case study with an application (RestoMapper) • No real evaluation	The authors created tests automatically from existing US-UIDs of an existing application using their approach. The resulting test set was compared to an existing test set that was created manually. The code of the application was manipulated using code mutation and lack of code. For the method code mutation the values of an array were changed manually. For lack of code a class was deleted.
4b	The approach can be applied on an example. Otherwise no evaluation results because the authors state that an evaluation is beyond the limitations of their work. The reason for this is that the quality of the created tests in this approach highly depends on the experience and skill of the person executing the approach.	Code mutation and lack of code resulted for both test sets in failed tests. Lack of code also resulted in not executable tests. The authors concluded that the tests created by their approach can successfully classify tests as successful, failed or not executable.

Both approaches provide a possible way to create acceptance tests that are executable with the tool *FitNesse*. El-Attar and Smith utilize use case models and domain models as their initial data while Longo et al. only need a non-specific description of the use cases. Generally El-Attar and Smith use more artifacts as their approach needs use case models, domain models, high-level acceptance tests and robustness diagrams as intermediate steps to the final representation of the executable acceptance tests. In the approach of Longo et al. only US-UIDs need to be created which are then automatically converted to *Fit-tables*. In contrary to El-Attar and Smith the approach of Longo et al. requires the creation of some code in the process: This is the case because the used *Fit-tables* differ between the two approaches. While El-Attar and Smith use the specific table types ActionFixture, RowFixture and ColumnFixture, Longo et al. use easier *Fit-tables* that are connected to the System-under-test via a Fixture-Class. This Fixture-Class has to be written manually.

Both approaches provide a tool to combine artifacts with the resulting acceptance tests which helps traceability. For both approaches the creation of the executable acceptance tests is still mostly or completely manual. While the approach of El-Attar and Smith uses no automation during the creation of the executable acceptance tests, the last step of the approach of Longo et al. is fully automatically. This is possible because the US-UIDs created in the approach of Longo et al. are a different way to display the information of a *Fit-table* and therefore can be directly converted to a *Fit-table*. The execution of the final acceptance tests is fully automatic for both approaches.

Both approaches involve customer and developers as stakeholders. The customer delivers the User Scenarios and receives (because of the development of acceptance tests) potentially a final product that is closer to his needs. The developers receive automatically executable tests that help them during the development process to determine which requirements are already satisfied and which still need to be implemented. While in the approach of Longo et al. the creation process of the acceptance tests is done by the customer and the developers together, in the approach of El-Attar and Smith a Business Analyst is responsible for this process.

El-Attar and Smith only visualize their approach through an example and state that the approach cannot be validated in their work because it is beyond the limitations of their work. The reason for this is that all the steps to create the acceptance tests are done manually and

therefore depend on the experience and skill of the analyst performing the steps. Longo et al. include a small evaluation in their work. They compare the tests created by their approach to tests that are created without guidelines from the same US-UIDs. During the testing phase they conclude that the tests created by their approach can be classified as *successful*, *failed* or *not executable*. Also changes in the source code of the System-under-test resulted in failed tests for both of the test sets which leads the authors to the conclusion that both the tests created without guidelines as well as the tests created with their approach can detect fails in the System-under-test.

2.6 Conclusion

The literature search showed that creating acceptance tests that are automatically executable with the specific tool *FitNesse* is not a widely researched topic in the literature. However, approaches exist that differ in their process to create tests. Two of these approaches were presented in this chapter:

The approach by El-Attar & Smith is aimed at larger projects and therefore, might not be useful for smaller products. It requires the creation of a lot of UML models. If an analyst exists that has experience in creating these models and at least a few of the used models are created anyway in the engineering process, then this approach might be useful.

The second approach by Longo et al. could be used for smaller projects where the customer is heavily involved. The customers have to be involved because they have to provide the User Scenarios in this approach. The approach is heavily dependent on the creation of US-UIDs that contain the information of Fit-tables in a different (possibly better) way. If the developers and customers prefer US-UIDs over Fit-tables and they want to use User Scenarios, this approach might be useful.

Overall, in the considered approaches the creation of acceptance tests is a process that is highly dependent on the experience and skill of the persons involved. Once the executable tests are created they are an easy way to measure how well the requirements of the customer are implemented.

3 Testing with a transition system

3.1 Introduction

System tests are used to make sure that clients receive exactly the kind of software they previously specified within the order contract submitted to the software vendor. Oftentimes what was specified and what was implemented does not match entirely in the end. The software vendor faces traceability problems to track which requirements could be covered in which tests and therefore which requirements got implemented. How does one make sure that each atomic functional and robustness requirement specified is covered in the requirements' implementation?

To solve this, the process of formal definition of requirements and matching system tests must be brought closer together. Testing should already be possible in early stages of development, to be precise during the specification phase already. To automatically derive test scenarios from means of the specification area *transition systems* are used. They help to generate test paths as possible combinations of fine-grained functional requirements received through the traversal. An equivalent approach can be used to derive robustness tests as well. In this process requirements can furthermore be tested on their consistency, correctness and integration and eventually can be refined further.

For this purpose the two approaches [47] and [50] were analyzed. While [47] was given in advance by the advisors, [50] was discovered through an extensive literature search shown in section 3.2. Both approaches will be explained in the following sections section 3.3 and section 3.4 and applied to the movie management software example. A comparison between the two approaches will be drawn using a synthesis matrix shown in section 3.5. The main results of testing with transitions systems will be summarized in section 3.6.

Please refer to the glossary in order to receive a common understanding of the following terms used in the sections below: contract, coverage criterion, interaction overview diagram, object constraint language, operation, test case, test objective, test scenario, transition system, UC-System, UC-SCSystem, use case, use case scenario, XML metadata interchange.

3.2 Literature search

The literature research was driven by the central research question (RQ):

„Which approaches for automatic generation of system tests exist that are using contract enriched use cases or other use case related means of the specification area within a transition system simulation model?“

The focus during this literature search was on finding a second approach to automatically generate system tests from means of the specification phase by exhaustively simulating a transition

system to generate test paths similar to [47]. But as [47] is restricted on using *contract* enriched *use cases* and *use case scenarios*, the way how *test objectives* are derived was this time freely selectable to receive another new, but similar approach. The pre-search results were promising both on IEEE Xplore ([34]) and ACM ([3]). Only some papers on ACM could not be accessed publicly. The number of results was considered to be sufficient to cover all relevant scientific papers, which is why the research was restricted on these two platforms. Furthermore, two relevance criteria inspired by the central RQ were defined:

- Does the method described in the article generate system tests automatically from use cases or other use case related means of the specification area?

- Are test objectives generated using some kind of simulation model based on use case contracts (pre- and postconditions) or similar transition system approaches?

As system tests can not exclusively be derived from means of the specification area, an article should restrict to generating system tests from use case related means of the specification area. To derive test objectives a transition system should be simulated exhaustively based on contract definitions (pre- and postconditions).

The search was done using both snowballing and search term techniques. 140 papers were found referencing [47] and 46 articles were referenced by [47]. The results of the snowballing search can be found in Table 3.1. The backward snowballing approach was restricted on references stated in the *Related Work* chapter as all other references relate to preceding work that serve as basic knowledge to realize the transition system approach in [47]. Additionally, most of the references were quite old since the original paper was published in 2006. Therefore not all papers could be found on IEEE Xplore or ACM, which is why the number of considered papers is even lower than the number of original references found. Why specific papers were considered not suitable or only partly suitable is documented in [31].

Table 3.1: Results of snowballing techniques

	Yes	**Possibly**	**No**
Forward snowballing	5	9	63
Backward snowballing	2	1	2

Seach-term based search was done using the following key terms: system tests, automatic generation, transition system, simulation model, use cases, contracts. Only papers published between 2006 and 2020 having the search term “test” and “use case” in their publication title were evaluated.

Table 3.2: Results of search-term based technique

Source	**Date**	**Search restrictions**	**Search query**	**#Results**
IEEE Xplore	2020-11-21	"system tests" in document title; "automatic generation" in document title; "transition system" in full text & metadata; "simulation model" in full text & metadata; "use cases" in document title; "contracts" in full text & metadata;	"system tests" AND "automatic generation" AND "transition system" AND "simulation model" AND "use cases" AND contracts	0
IEEE Xplore	2020-11-21	"system tests" in document title; "automatic generation" in abstract; "transition system" in full text & metadata; "use cases" in document title; "contracts" in full text & metadata;	"system tests" AND "automatic generation" AND "transition system" AND "use cases" AND contracts	0
IEEE Xplore	2020-11-21	"test" in document title; "transition system" in full text & metadata; "use cases" in document title	"system tests" AND "transition system" AND "use cases"	82
ACM	2020-11-21	"system tests" in title; "automatic generation" in title; "transition system" in full text; "simulation model" in full text; "use cases" in title; "contracts" in full text;	"system tests" AND "automatic generation" AND "transition system" AND "simulation model" AND "use cases" AND contracts	0

ACM	2020-11-21	“system tests” in title; “automatic generation” in abstract; “transition system” in full text; “use cases” in title; “contracts” in full text;	“system tests” AND “automatic generation” AND “transition system” AND “use case” AND contracts	0
ACM	2020-11-21	“system tests” in title; “transition system” in full text; “use cases” in title	“system tests” AND “transition system” AND “use cases”	0
ACM	2020-11-21	“test” in title; “use case” in title	“system tests” AND “use case”	8

From the resulting papers, only one was considered suitable as a potential second paper. After the search for potential articles to be evaluated was finished, a choice between eight remaining papers from the initial search had to be made:

- System Testing using UML Models [51],
- An Automatic Tool for Generating Test Cases from the System's Requirements [33],
- Automated Test Case Generation from Use Case: A Model Based Approach [19],
- Requirements Document Based Test Scenario Generation for Web Application Scenario Testing [60],
- An Approach to Modeling and Testing Web Applications Based on Use Cases [40],
- Test cases generation from UML state diagrams [38],
- Requirements by Contracts allow Automated System Testing [46],
- An Automated Approach to System Testing Based on Scenarios and Operations Contracts [50].

The decision criteria are based on the different search terms mentioned above and the already defined criteria. Additionally, focus of the selected paper should lie on creating system tests for any generic application area, not just UI related parts of an application.

The paper *An Automatic Tool for Generating Test Cases from the System's Requirements* was not chosen as it does not focus on testing the consistency of use case combinations with contracts to build test objectives as in the original paper. Furthermore, it is not as in-depth as the original paper. Contract enriched use cases could neither be found in *System Testing using UML Models.*

Automated Test Case Generation from Use Case: A Model Based Approach really embodies the principle of state base modeling based on use cases with its *interaction finite automaton* (IFA), but doesn't introduce a formal language to define use cases and its contracts.

Requirements Document Based Test Scenario Generation for Web Application Scenario Testing as well as *An Approach to Modeling and Testing Web Applications Based on Use Cases* are specifically optimized for web application *test scenarios* and therefore not as general and universally applicable as the original paper.

Test cases generation from UML state diagrams and *Requirements by Contracts allow Automated System Testing* could unfortunately not be accessed in full length in IEEE Xplore.

The chosen article to further evaluate is *An Automated Approach to System Testing based on Scenarios and Operations Contracts*, as it introduces a second way to create system tests from use case scenarios as UML 2.0 models by enriching it with contracts and by transforming the formalized scenarios to a transition system to derive test objectives. It uses a more graphical approach to define use cases instead of using a formalized language to do so and goes deeper down into use-case level instead of system-level test generation. A more in-depth comparison between the two papers can be found in section 3.5.

3.3 Approach 1: Automatic test generation: A Use Case driven approach

3.3.1 Description

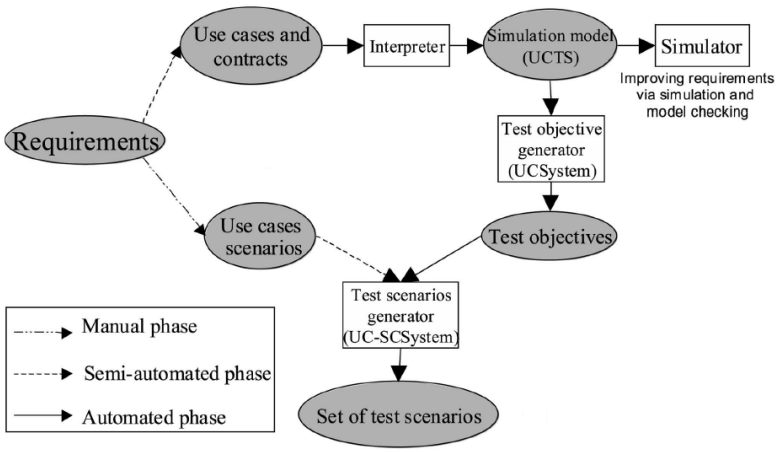

Figure 3.1: Flow of generating test scenarios [47]

The approach consists of two phases (see Figure 3.1). In the first phase, UML use cases get enhanced with contracts (pre- and postconditions). Use cases can be seen as the systems' main functions whereas contracts are used to infer the correct partial ordering of functionalities that the system should offer. They express the ordering constraints between the use cases during the simulation process to build the transition system. The contracts are made executable for logical evaluation by writing them in the form of requirement-level first-order logical expressions. They consist of predicates with and are used to describe facts in the system like actor state, main concept state, roles, and more. Each predicate can either evaluate to true or false, but never to undefined. A dedicated editor tool helps to manage the predicates and guides

the design of contracts to maintain a nonredundant, minimal set of contracts and predicates. Contracts therefore specify the system properties to make a use case applicable (preconditions) and define the system properties (predicates) acquired after their application (deterministic postconditions). Parameters to use cases can either be system actors or main concepts of the use case, which are represented as normal instantiated objects of classes during the test scenario generation process.

Through exhaustive simulation by the prototype/interpreter-tool *UC-System* a *use case transition system* (UCTS) is built, which serves as a model for all valid sequences of use cases. Therefore first an initial state and enumerations of different business entities that later serve as parameters to use cases are declared. Then all use cases get instantiated by replacing the set of formal parameters with all the possible combinations of their possible actual values (i.e. actors and main concepts). To apply an instantiated use case the simulation state and the precondition of an instantiated use case must match. The simulation state is then updated according to the postcondition of the contract. The initial state defines which predicates are true from the very beginning, the current simulation state covers all instantiated predicates which are evaluated to true. Now all possible paths are traversed exhaustively until a final UCTS was generated.

During this step, the requirement engineer has also the chance to check and possibly correct the requirements before the tests are generated. Inconsistencies between predicates and contracts can be identified as well as underspecification or errors in the requirements. Invariants can also be checked. The number of states in the transition system can be calculated with

$$maxsize_{UCTS} = 2^{n_{ip}} \tag{3.1}$$

where

$$n_{ip} = p \cdot max_{instances}^{max_{param}}. \tag{3.2}$$

n_{ip} describes the amount of instantiated predicates, p the number of predicates, $max_{instances}$ the maximum amount of instances for a parameter (actors and main concepts of the use case), and max_{param} the maximum amount of parameters per predicate p. In practice, many of the potential states are not reachable and only a small number of instances are necessary for achieving a proper statement coverage.

Now relevant test objectives get extracted from the UCTS by applying predefined *coverage criteria*.

The *All Edges* (AE) criterion makes sure that all state transitions are covered, whereas the *All Vertices* (AV) criterion guarantees that all states (predicates) are reached within the set of test objectives. The *All Instantiated Use Cases* (AIUC) criterion is helpful in case a state transition can be done by multiple use cases or a use case leads to no state change. A combination od AV and AIUC is the *All vertices and All Instantiated Use Cases* (AV-AIUC) criterion. The most strict criterion is probably the *All Precondition Terms* (APT) criterion, which makes sure that all possible ways to apply each use case are exercised. Now, these criteria are mainly introduced to generate functional tests. The *Robustness* criterion on the other hand explicitly exercises a use case in as many different ways as to make its precondition false. Therefore valid test paths are generated, which lead to an invalid application of a use case to generate robustness tests from. All algorithms are based on breadth-first search in the UCTS to obtain small test-objectives that are human-understandable and meaningful.

Subsequently, in the second phase, test scenarios get generated by replacing each use case in a test objective with the according use case scenario that is compatible in terms of static contract matching. This is done by the prototype-tool *UC-SCSystem*. The use case scenarios were also attached with contracts beforehand. This time the contracts contain more detailed pre- and postconditions. There are contracts that rely on the rest of the model, they are written in *Object Constraint Language* (OCL), and there are contracts that rely on the predicates of the use cases. Now the exchange of messages involved between the environment and the system is also specified. Note that all use case scenarios are system-level scenarios. Eventually, additional parameters and messages need to be passed manually before executable *test cases* can be generated. The process results in executable test scenarios that get evaluated using the statement coverage metric.

One possible challenge of the approach is that the simulation model has to be compact enough to avoid combinatorial explosion of the internal states. Therefore the two-phase approach was chosen and parameters to instantiate use cases during the simulation can often be restricted to only the main system concepts and actors. Furthermore, the above-mentioned test objective generation criteria were identified through experimental comparisons and help to keep the number of test objectives in a reasonable scope.

The generated test scenarios can either lead to a pass verdict, a fail verdict (in case a postcondition is violated), or an inconclusive verdict. The latter is invoked if a precondition is evaluated to false and the test scenario was not executed entirely. This could be because of underspecification or because of inappropriate test data. To solve this either a new initial state (test data) has to be defined or additional test cases that execute the remaining use case scenarios need to be provided.

To evaluate the approach the original authors used three software products: An Automated Teller Machine (ATM) with 850 lines of code, an FTP server with 500 lines of code, and a virtual meeting (VM) server with 2.500 lines of code. Statistics on the amount of generated test cases can be found in Table 3.3.

Table 3.3: Statictics of the generated test cases

	ATM	**FTP**	**VM**
# use cases	5	14	14
# nominal UC-scenarios	5	14	14
# exceptional UC-scenarios	17	14	14
# generated functional test cases	6	14	15
# generated robustness test cases	17	33	65

Taken the example of the VM server, most coverage criteria reached up to 70% code coverage, when including robustness test cases even up to 80%. For more detailed information see Table 3.4.

All coverage criteria are almost equal in their achieved code coverage, with the exception of the AV criterion. Here the code coverage is low as not all use cases can be covered, especially those use cases that do not change the system state are missing. The ratio between the covered statements and the amount of generated test cases gives information about the efficiency of the generated test scenarios. Here the AIUC and APT criteria scored best. The APT criterion manages to reach 100% functional code coverage with only 15 test cases. The efficiency of the robustness criterion on the other hand scored quite low, 65 test cases could only cover up to

Table 3.4: Statement coverage reached by the generated test cases

	ATM	FTP	VM
% of functional code covered	100%	90.7%	100%
% of robustness wrt. the spec covered	42.31%	38.6%	52%
% of code covered (total)	94.76%	72.5%	80%

50% of the equivalent robustness code. Therefore the approach works well for functional code, but not so well for robustness code. This is because only violations of the use case attached preconditions are taken into account, inappropriate test data, or violating the more detailed preconditions of use case scenarios are not included in the test generation process.

One possible extension of the approach was also considered. Activity diagrams could be chosen to model the use case dependencies in a more graphical approach, which is then shown in the upcoming approach two.

3.3.2 Application

In the first step, the test objectives have to be derived. Therefore we define the use cases and their contracts (Listing 3.1) as requirement-level first-order logical expressions. The contracts are used to infer the correct partial ordering of functionalities that the system should offer. Only the use cases that really impact the state of the transition system were specified for this example. The notation used is equal to the one proposed in the paper. *UC* introduces the identifier and parameters of a use case, *pre* marks the beginning of the precondition logical expression and *post* the beginning of the postcondition logical expression. Furthermore, logical operators like *and* and *or*, quantifiers like *forall* and *exists*, and implications by using *implies* can be used. The expression *@pre* ensures that a given logical expression has already been evaluated to true when evaluating the precondition.

Listing 3.1: Contracts attached to use cases

```
1
2   UC createMovie(m: movie)
3   post createdMovie(m)
4
5   UC createLinkedPerformer(p: performer, m: movie)
6   pre createdMovie(m)
7   post createdPerformer(p) and createdLink(p,m)
8
9   UC rateMovie(m: movie)
10  pre createdMovie(m)
11  post calculatedOverallRating(m)
12
13  UC ratePerformer(p: performer)
14  pre createdPerformer(p)
15  post forall(m: movie){ createdLink(p,m)@pre implies calculatedOverallRating
     (m) }
16
17  UC linkExistingMovie(m: movie, p: performer)
18  pre createdMovie(m) and createdPerformer(p)
```

```
19    post not createdLink(p,m)@pre implies (createdLink(p,m) and
       calculatedOverallRating(m))
20
21    UC linkExistingPerformer(m: movie, p: performer)
22    pre createdMovie(m) and createdPerformer(p)
23    post not createdLink(p,m)@pre implies (createdLink(p,m) and
       calculatedOverallRating(m))
24
25    UC unlinkMovie(m: movie, p: performer)
26    pre createdMovie(m) and createdPerformer(p) and createdLink(p,m)
27    post calculatedOverallRating(m) and not createdLink(p,m) and not exists(m2:
        movie){ createdLink(p,m2) }@pre implies not createdPerformer(p)
28
29    UC unlinkPerformer(m: movie, p: performer)
30    pre createdMovie(m) and createdPerformer(p) and createdLink(p,m)
31    post calculatedOverallRating(m) and not createdLink(p,m) and not exists(m2:
        movie){ createdLink(p,m2) }@pre implies not createdPerformer(p)
32
33    UC removeMovie(m: movie)
34    pre createdMovie(m)
35    post not createdMovie(m) and forall(p: performer){ not createdLink(p,m) }
       and not exist(m2: movie){ createdLink(p,m2) }@pre implies not
       createdPerformer(p)
36
37    UC removePerformer(p: performer)
38    pre createdPerformer(p)
39    post not createdPerformer(p) and forall(m: movie){ not createdLink(p,m) and
        calculatedOverallRating(m) }
```

After that, the UC-System prototype/interpreter tool should build the UCTS (Figure 3.2) through exhaustive simulation. The pool of parameters was restricted to one movie and performer to avoid a combinatorial explosion for this example. To build instantiated use cases the set of formal parameters are replaced with all the possible combinations of their actual values. In our case, we use the most simple approach by just having one possible combination. Furthermore, the predicate calculatedOverallRating is no longer considered. Note that only predicates that evaluate to true are listed in the states as in the original paper.

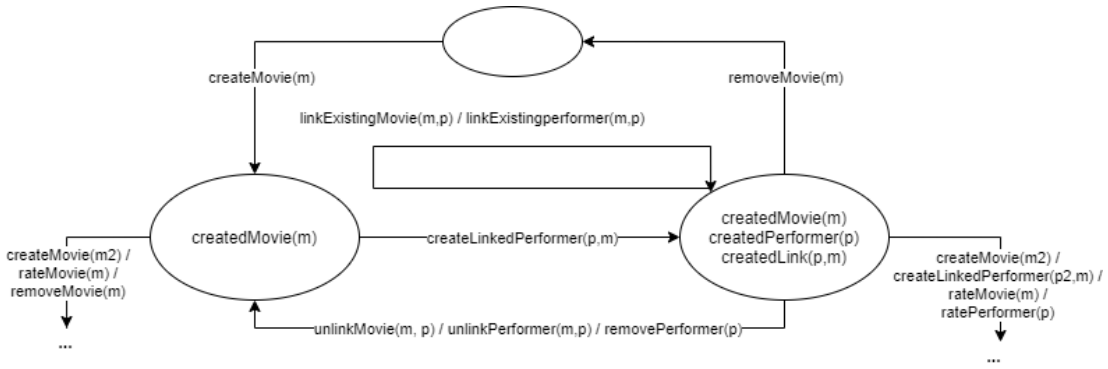

Figure 3.2: The use case transition system

After applying an instantiated use case in the transition system (in case the precondition of the contract was fulfilled) the simulation state is updated according to the contracts' postcondition.

Depending on the selected coverage criterion, we receive different test objectives as correct sequences of use cases. The robustness criterion was not considered in this example, but its application is coherent to the functional coverage criteria. How many test objectives are derived depends on the internal implementation of UC-System and cannot be predicted for this example. Let's assume that one test objective is the test path createMovie(m) -> createLinkedPerformer(p,m) -> removeMovie(m). Then the use case scenarios from Figure 3.3 are used to replace the use cases in the test objectives. It helps to specify the exchange of messages involved between the environment and the system.

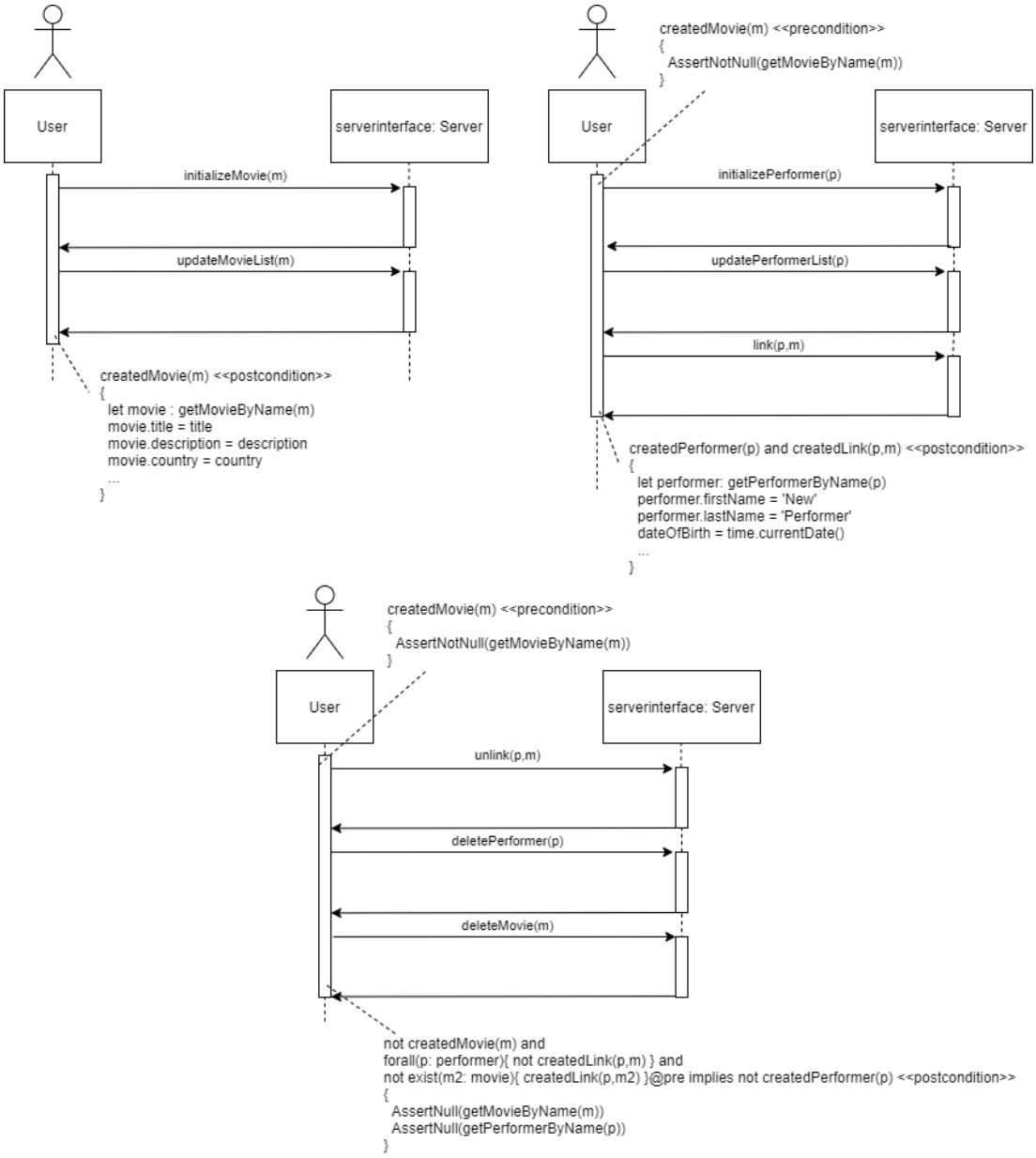

Figure 3.3: The use case scenarios

Note that the use case scenarios may still be incomplete for the execution. They contain the main messages exchanged between the tester and the SUT and say how the system has to be simulated to perform a use case and how to react to the simulation. To know how the system has to be simulated, the use case scenarios contain more detailed contracts written in OCL besides the contracts written as logical expressions that were provided by the use cases.

The prototype-tool UC-SCSystem uses the shown implementation in the use case scenarios to derive executable test scenarios as JUnit tests.

3.4 Approach 2: An Automated Approach to System Testing based on Scenarios and Operations Contracts

3.4.1 Description

Based on the suggested improvements in the first paper, the second paper uses a more graphical approach using *interaction overview diagrams* (IOD), a special form of activity diagram used to show control flow, to derive test paths. It helps to start testing in the early stages of development. Each node in the IOD represents either an interaction diagram (sequence diagrams) or interaction occurrences that show an *operation* invocation. Every IOD corresponds to one use case. The IODs get enhanced with contracts written in the *object constraint language* (OCL) and they get transformed into a contracts transition system (CTS) which models all scenarios of the IOD. Here the states are represented by the contracts and the transitions by the operations (interaction diagrams or interaction occurrences) in the IOD. The CTS is built by a tool using the *XML metadata interchange* (XMI) file for the IOD and the operation contracts in OCL as inputs. A state is created against each precondition and each postcondition of the operations. Logical if-then-else conditions are resolved by combining their testing condition with the result, therefore two different sub-states are created. The surrounding postcondition is then a composition of the two sub-states. Additional transitions are added for all conditional flows leading to alternative scenarios and their guard conditions are attached to them. Additional CTS flows help to further refine the requirements by spotting potential unwanted behavior or underspecification. The CTS is often visualized in a matrix.

Through traversing the CTS test paths get derived. Therefore different coverage criteria are defined. The simplest coverage criterion is the *state coverage* criterion, which generates test paths until all states of the CTS are covered. The criterion is already covered by the wider *transition coverage* criterion, which makes sure that all transitions are traversed before stopping the test path generation. The most expensive criterion is the *transition pair coverage* criterion, each possible transition pair needs to be covered in the test path generation process. It was detected, that the transition coverage criterion delivers a reasonable amount of test paths, but is not suitable for fault detection every time (compared to the transition pair coverage criterion which scores best in this task).

Besides using a graphical approach with IODs instead of using a formal requirement level language, the key difference to the first paper is that test scenarios do not get generated on system-level but rather on use case level due to the fact that contracts are not attached to use cases (which can be compared to a complete IOD) but rather to all operations within the IODs. It therefore serves as a platform to generate more in-depth test scenarios (as well as for negative test cases). The paper does not provide additional steps on how to generate executable test scenarios from the retrieved test paths.

3.4.2 Application

The second approach differs from the first one as this time a transition system is built on a concrete use case, in our case the use case to unlink a movie from a performer. Input to the approach in this paper is the IOD (Figure 3.4) with separate contracts defined in an OCL file (Listing 3.2). IODs are a special form of activity diagrams used to show the control flow. The nodes in our case are UML sequence diagrams and define the operations of the CTS. The states are represented by the contracts themselves.

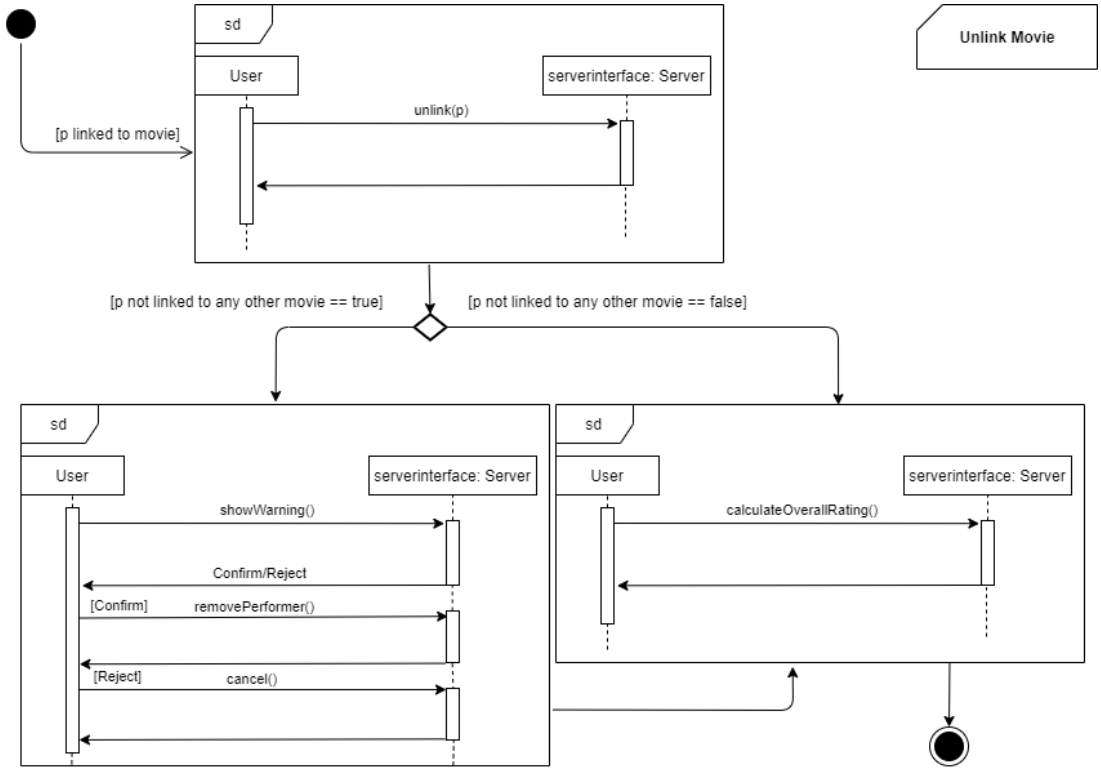

Figure 3.4: The interaction overview diagram

Listing 3.2: Contracts written in OCL

```
context Movie::unlink(performer)
pre  self.performers[performer] -> not isEmpty()
post self.performers[performer] -> isEmpty()

context MovieManager::removePerformer(performer)
pre forAll(movie | movie.performers[performer] -> isEmpty())
post self.performers[performer] -> isEmpty()

context Movie::calculateOverallRating()
post self.overallRating = 0.5 * (self.mean(self.performers.getRatings()) +
 self.rating)
```

Based on the IOD and the specified contracts the CTS matrix gets defined and leads to the CTS shown in Figure 3.5.

Operations	Pre	Post	Composite States
O_1	S_0	S_1 OR S_2	A
O_2	S_1	S_3 OR S_4	B
O_3	S_3 OR S_4	S_2	
O_3	S_2	S_5	

The operations (sequence diagrams in the IOD) can be thought of as the transitions in the CTS, whereas the states match the specified postconditions (maybe thought of as starting points of arrows).

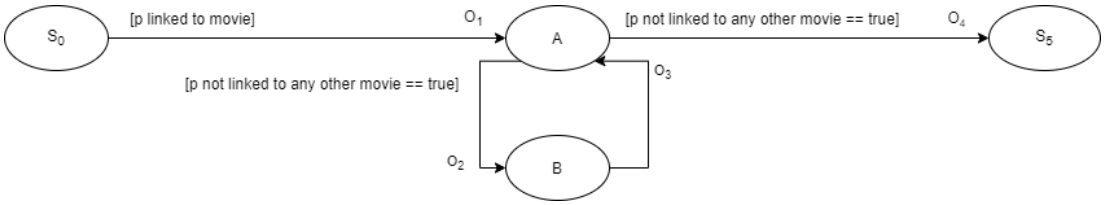

Figure 3.5: Contracts Transition System

For instance, the state S_0 describes the state when a performer p is linked to the given movie. One can then apply the operation O_1 to unlink the performer, resulting in the composite state A where either the performer is not linked to any other movie S_1 or the performer is still linked to another movie S_2.

Based on a coverage criterion the test paths get derived. Different from the first approach no test scenarios get generated. The paper only shows a new more low-level, graphical approach to generate test paths as this was even a suggested improvement from the authors of the first paper.

3.5 Comparison

Table 3.6: Synthesis matrix

No.	Approach 1	Approach 2
1a)	Use cases describing the basic operations in the transition system. Contracts that are attached to the use cases describing the states in the transition system. A contract consists of pre- and postconditions that specify the system properties to make a use case applicable and which properties are acquired by the system after its application. Parameters to contracts are actors and main concepts of the use case. The transition system (UCTS) itself as a simulation model to derive test objectives. The states in the UCTS are given through the predicates defined in the contracts, the transitions are triggered when applying an instantiated use case. Test objectives describing the test paths. Test objectives are derived by traversing the UCTS using a specific coverage criteria. Use case scenarios to build test scenarios from test objectives. Use case scenarios contain the main messages exchanged between the tester and the SUT, they define how the system has to be simulated to perform a use case and how to react to the simulation.	IODs holding all scenarios and operations of a use case. Operations can either be interaction uses or sequence diagrams inside the IOD and define the state transitions. Contracts written in OCL that are attached to the operations describing the states in the transition system. The CTS describing the transition system. Test paths derived from the CTS using coverage criteria.
1b)	Use cases, contracts written as logical expressions, use case scenarios (sequence diagrams), initial system state, selected coverage criterion, and additional use case scenario parameters.	IODs, contracts written in OCL, selected coverage criterion, possibly manual resolving of conflicts in the CTS matrix.

1c)	To express the ordering constraints between use cases, each use case is attached by a contract. To simulate the use cases, the set of formal parameters of the contracts are replaced with all possible combinations of their actual values. The use cases are then called *instantiated*. To apply an instantiated use case the precondition of its contract must match with the current simulation state. Afterwards the simulation state is updated according to the postcondition of the use case. Exhaustively simulating the system results in the UCTS. To derive test objectives the transition system is traversed according to one of the predefined coverage criteria AE, AV, AIUC, AV-AIUC, or APT. To build test scenarios a use case scenario can replace a use case at a certain stage of execution if the state reached at this stage locally implies the precondition of the use case scenario. Executable test scenarios are generated by the prototype-tool UC-SCSystem.	Each operation in the IOD was enriched with its own contract written in OCL. To build the CTS, first all operations have to be identified from the IOD. The operations are taken from the sequence diagrams or from other operations expressed as interaction occurrences in the IOD. Using the contracts of operations, the states for the CTS are identified. Eventually, conflicts in the CTS have to be resolved such as logical if-then-else conditions, equal contract statements, or join nodes. After the CTS was built, test paths are derived from the CTS by applying a coverage criterion, which is either state-, transition- or transition pair coverage.
2a)	Automatic test generation from use cases and use case scenarios. Requirement validation by identifying inconsistencies, underspecifications, and invariants.	Deriving test paths from IODs. Further requirement validation on use case level.
2b)	Test writers / Developers, Requirement Engineers.	Test writers / Developers, Requirement Engineers.
2c)	Software Requirements (definition of a software requirement, functional requirements, acceptance tests), Software Testing (model-based techniques, objectives of testing, evaluation of the tests performed), Software Engineering Models and Methods (preconditions, postconditions and invariants, behavioral modeling, analysis for consistency, and correctness, traceability).	Software Requirements (definition of a software requirement, functional requirements, acceptance tests), Software Testing (model-based techniques), Software Engineering Models and Methods (preconditions, postconditions and invariants, behavioral modeling, analysis for consistency and correctness).
3a)	Dedicated editor to design use cases with contracts, UC-System to build the UCTS simulation model, and to derive test objectives from it. UC-SCSystem to exchange the use cases by use case scenarios in order to build the executable test case scenarios.	UML 2.0 as a standard for IODs, OCL as a formal language to write the contracts, prototype tool to derive test paths.
3b)	Writing the use cases and contracts is supported by a dedicated editor, but has to be done manually. Deriving test objectives from use cases and contracts through the transition system is done automatically by UC-System. Use case scenarios have to be specified manually, the generation of test scenarios works semi-automatically with UC-SCSystem as it may need additional parameters from the tester.	Only the IOD and contract specification has to be done manually, the complete approach was then automized by a prototype tool.

4a)	The approach was evaluated by looking at the statement coverage of three sample programs and the efficiency of test case scenario generation.	The approach was evaluated by looking at the number of test paths generated to cover all success scenarios and fault detections.
4b)	Code coverage with the most coverage criteria was around 80%. The coverage criteria differ in efficiency. AE, AV, and AV-AIUC perform with low efficiency, the sets of test cases are larger than in AIUC and APT. APT reaches 100% functional test coverage with only 15 test case scenarios. Testing robustness leads to a high number of generated test case scenarios that only cover about 50% of the corresponding code. The approach is good for functional testing, but bad for robustness testing.	Using the transition criterion to derive test paths leads to a reasonable amount of test paths and covers all alternative flows in the IOD, but is not suitable for fault detection at any time. The transition pair coverage criterion guarantees the maximum fault detection, but leads to a high amount of test paths. State coverage captures all success scenarios.

Both approaches have a transition system as a core component in common. Both use contracts to define the states in the transition system. A transition can be applied in case the precondition is fulfilled, the state is then updated according to the postcondition. The transition system is used to derive test paths. Furthermore, both approaches define coverage criteria to generate a certain amount of test cases from the transition system. Both have the AE/transition coverage criterion and AV/state coverage criterion in common. Using this core idea it is also easy in both approaches for requirement engineers to validate their specified requirements during the simulation of the transition system. Both papers draw a similar conclusion, that the core approach is helpful to generate functional test case scenarios for the requirements, nevertheless, there is a lack to achieve a comparable coverage for robustness code for fault detection.

Nevertheless, the two approaches differ in their case of application. While [47] uses use cases to automatically generate system-level tests, [50] restricts to generate test paths for a specific use case. It therefore does not need additional use case scenarios as an input like [47] to generate test scenarios as these are already natively given in the IOD.

Besides the case of application, the input parameters in both approaches are different as well. [47] uses requirement-level first-order logical expressions to describe use cases and their contracts. A more graphical approach was chosen in [50] by specifying a use case with an IOD instead of using a formal language.

[47] has the special characteristic that it demonstrates an almost fully automized end-to-end test scenario generation process until concrete test execution, which is missing in [50]. Nevertheless, this could easily be implemented in [50] as well using similar tool support as in [47].

3.6 Conclusion

Testing with a transition system delivers a method on how to bring requirement specification and system test generation closer together, eliminating traceability problems between what was specified and what was implemented. Requirements can easily be validated on their consistency, correctness, and on possible underspecification while at the same time test paths can be derived through traversing the transition system.

In literature search, a second paper besides the given one was found that follows the process of automatically generating test paths through traversing a transition system, but this time uses the in [47] proposed extension of IODs as a form of activity diagrams instead of use cases as its input besides contracts. To find this article both snowballing and search-term-based techniques were used, whereas the choice of relevant articles was based on previously specified relevance criteria. Eight resulting papers were found according to these criteria, [50] was finally chosen.

Approach [47] almost shows a fully automated way to derive executable test scenarios as JUnit tests. Contract enriched use cases using a formal language based on requirement-level first-order logical expressions are used as an input to the simulation model. Through exhaustive simulation, a transition system is build from which test objectives are derived using coverage criteria. It was empirically proven that the AIUC and APT criterion perform the most efficient and the most extensive measured by statement coverage. Giving contract enriched use case scenarios as a second input helps to generate concrete executable test scenarios.

[50] switches from system level to use case level test generation. It uses contract enriched IODs as graphical input and builds a transition system on it. Test paths are generated using the transition coverage criterion. No further methodology was introduced to generate executable test cases, but as the approach is already specified on use case level, the generation of test code is self-explanatory (a similar tool like UC-SCSystem from [47] can be used).

One weakness of both approaches remains the capability to generate a sufficient amount of robustness tests for fault detection. Both approaches only rely on contract violations and do not sufficiently cover data variations as test path generation is only based on one initial state chosen by the test creator. Still, for the automatic generation of functional test scenarios, both approaches show industry-standard and highly applicable solutions.

4 Testing with a timing component

4.1 Introduction

Requirements are the basis for all testing. In practice, requirements are often available in a wide variety of formats: As natural language text, spreadsheets, UML diagrams etc.... In addition to that, requirements are often not close to testing. In the course of the development phases of a program, the requirements may expand or change. In industry, individual software components are often developed by different manufacturers before they are combined into a system by the client of these productions. Although components are tested individually and independently, many errors can only be discovered when the components are integrated. Therefore Test cases and requirements must be more closely linked. Flaws and vagueness in requirements are common and hard to discover. "On account of this testing time may be wasted with test cases that result in the detection of flaws in the requirements and can thus not be used to evaluate the system under test (SUT)" [53]. Therefore requirements must be stricter and more formal.
Furthermore, there are systems that have to fulfil real-time requirements. Two approaches (Chapter 4.3 and 4.4) will be presented in this chapter. Both approaches test with a time component as they refer to a real-time system. In these systems, it is not only important what input the individual system functions receive (for example, through a boundary value analysis), but also how long they take. The same input in a real-time system can lead to a different system reaction if the execution time differs. But not only the execution duration of a system function, also the time between the execution of a system function can result in another system reaction. For this reason, the temporal aspect plays a major role in the approaches.

The first approach with the title "Model Based Requirements Analysis and Testing with Timed Usage Models (TUM)" is a graphically representation of requirements specification. "During the creation of the model the requirements are analysed and brought into an unambiguous and formal representation" [53]. Test cases are automatically generated from the graph using EXAM (Extended Automation Method). The second approach, "A Model-Based Testing Technique for Component-Based Real-Time Embedded Systems" (in addition to the formal specification of the requirements by models), deals even more specifically with the dependency between individual components of SUT, so that they are taken into account in the automatic test generation from these models.

Chapter 4.2 explains the process of the literature search that was used to find the second approach. Chapter 4.5 compares the two approaches and shows the result in a synthesis matrix. Chapter 4.6 provides an assessment of the approaches and summarises the most important findings.
Please refer to the glossary in order to receive a common understanding of the following terms used in the sections below: CACC, TUM, EXAM, CREMTEG, CIIG, CSIBG, CSIEDBG, AIC, AITC, AIEC.

4.2 Literature search

The literature search was conducted in too ways: Search term and snowballing. To find more and possibly different approaches the search question was chosen to be: “What models with a timing component exist that favor the creation of automatically generated tests?” The ACM [3] and IEEE [34] search sources were considered. Relevance criteria, both substantive and formal, were used to limit the number of search hits.

- Content criteria: “requirements as a core concept”, “support for automatically generated tests”, “model with consideration of timing”.

The content criteria were first searched for throughout the document. Since the number of hits was too large, the search of the content criteria was limited to the abstract of the article.

- Formal criteria: time criterion, diversity of author and language of article in English.

The time criterion was initially chosen so that only articles that were not older than ten years (publication date not less than 2010) were considered. Since the number of hits was still too large, the time criterion was finally set to 2015. The difference of the author from the given article should ensure that a second independent, not subjectively biased, scientific opinion on the topic “Testing with a time component” could be found.
Forward Snowballing produced 13 results, two of which were relevant (according to the criteria). Backward Snowballing also produced 13 results, none of which were relevant. All of the results did not meet the time criterion, as the given article was from 2010. Table 1.1 shows an overview of the search-term-based literature search.

Table 4.1: Overview of the search-term-based literature search

Search platform	Search date	Search restrictions	Search terms	# results	# relevant results	# relevant new results	Used articles	Comments
IEEE	18.11.2020		test AND requirements AND "Timed Usage Model"	2	0			too few results
IEEE	18.11.2020		test AND requirements AND model AND timing	385				too many results
IEEE	18.11.2020	Only articles whose constitutional datum is not older than 2010; Search query only related to "Abstract	"Abstract":test AND requirements AND model AND timing	156				still too many results
IEEE	18.11.2020	Only articles whose constitutional datum is not older than 2015; Search query only related to "abstract"; authors of the given article are not included in the search	"Abstract":test AND requirements AND model AND timing AND NOT "Authors":Sebastian Siegl AND NOT "Authors":Kai-Steffen Hielscher AND NOT "Authors":Reinhard German	81	5		Jing Guan, 2015: A model-based testing technique for component-based real-time embedded systems	first 50 hits were considered; Result hit set corresponds to expectations.
ACM	18.11.2020	Only articles whose constitutional datum is not older than 2015; Search query only related to "abstract"; authors of the given article are not included in the search	Abstract:(test AND requirements AND model AND timing) AND ContribAuthor:(NOT Sebastian Siegl) AND ContribAuthor:(NOT Kai-Steffen Hielscher) AND ContribAuthor:(NOT Reinhard German)	365				too many results

The search term in the first literature search method used was initially too specific (test AND requirements AND “Timed Usage Model”). This was generalised in the next step to increase the number of hits in the search query: “test AND requirements AND model AND timing”. This type of search was able to optimally fulfill the content criteria and the formal criteria. In combination with the previously mentioned criteria and the further restriction of the search that the search term only refers to the abstract of the article, the literature on which approach 2 is based was selected.

4.3 Approach 1: Model Based Requirements Analysis and Testing with Timed Usage Models

4.3.1 Description

In practice, requirements often exist in a wide variety of formats. A stricter formal notation of the requirements can be achieved through the so-called TUM (Timed Usage Model). During the creation of the model, the requirements are analysed and brought into a uniform form. Each path in the model is ultimately based on a requirement: in this way, the relationship between the previously created requirements and the later model-based requirements can be traced at any time. Each requirement must be simultaneously retrievable in a "path" of the TUM. The expected system reaction, after a requirement has been executed, must be marked in the diagram by means of an (end-) state. "Design" errors can be detected early in the creation of the model. The special feature of the model is the consideration of the time aspect: The states and state transitions in the model are each assigned a probability density function (pdf), which calculates the time how long the system remains in a certain state or how long the execution of an operation or a response of the SUT takes. Each state transition is therefore assigned a stimulus, which represents an operation of the SUT or a system response.

Figure 4.1 shows an easy example for a TUM: "S" are states, "a" stands for state transition, "t" describes a pdf either with respect to a state or to a transition, and "p" describes the transition probability from one state to the next.

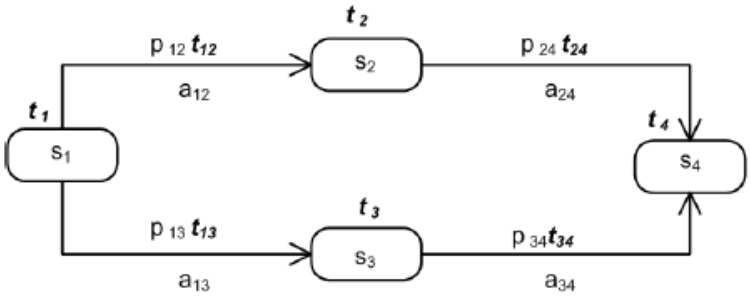

Figure 4.1: TUM [53]

The model serves as the basis for the entire testing process. It is used in the testing of real-time systems, e.g. in the automotive industry or in energy management. The model is created manually using a specially developed editor.
It is done in two steps: First, the system boundary will be defined. Second, all sequences of stimuli and their responses across the system boundary will be enumerated [53]. All stimuli that go across this boundary are extracted from the existing requirements [53]. Each possible sequence of stimuli will be mapped to a response of the system. A response can be composed of one or more outputs. "Two sequences u and v are equivalent if extended by any sequence w the future response is the same to uw and vw" [53]. These equivalent sequences are reduced to one sequence, since testing the other sequences would be redundant. In order to maintain an overview of the large number of all possible sequences, they are listed in order of length: It starts with an empty sequence λ and all sequences of length one are created first, then all sequences of length two and so on. By this procedure a complete and consistent usage model is created. The finished TUM is passed to EXAM in the next step. "EXAM comprises the

automated generation of platform dependent code and the automated execution of the derived test suite without human interactions" [53]. Test cases are automatically generated by running through the different paths of the TUM. All paths from the start to the final state are valid test cases.
For the test oracle, an appropriate counterpart from the EXAM test automation library is assigned to each individual stimulus. An importer allows thereby the import of a requirements library from a document-based requirements management system. The elements of this system can be traced back to each individual object in the TUM. The reference values or the expected result of one "test path" for the test oracle can be derived in three different ways: First automatically generated by computation rules, second by a textual description (program code or natural language) and third by measurements and checks on test benches.

4.3.2 Application

The approach "Model Based Requirements Analysis and Testing with Timed Usage Models" will now be applied to the Movie Manager, a collaborative project of students. The program comes with a domain diagram, an overview of all system functions, and a user task sheet about movie management in natural language. This data was used in the creation of the TUM. Data on the transition probability from one state to the next state is not available. This is therefore considered to be identical for all state transitions. In order to nevertheless realise a temporal component in the Movie Manager, the requirements were extended in the following way: Within the subtasks "Describe a movie" and "Describe a performer", changes were made to the system functions "show movie in IMDB" and "show performer in IMDB": After the error message "Connection failed" appears, the system should automatically try to establish a connection to the IMDB website [35]. If this exceeds a certain time value (in this case 120 sec.), a final message is issued to the user stating that the connection could not be re-established automatically. After this message appears, no further attempts are made to connect automatically to the IMDB server.

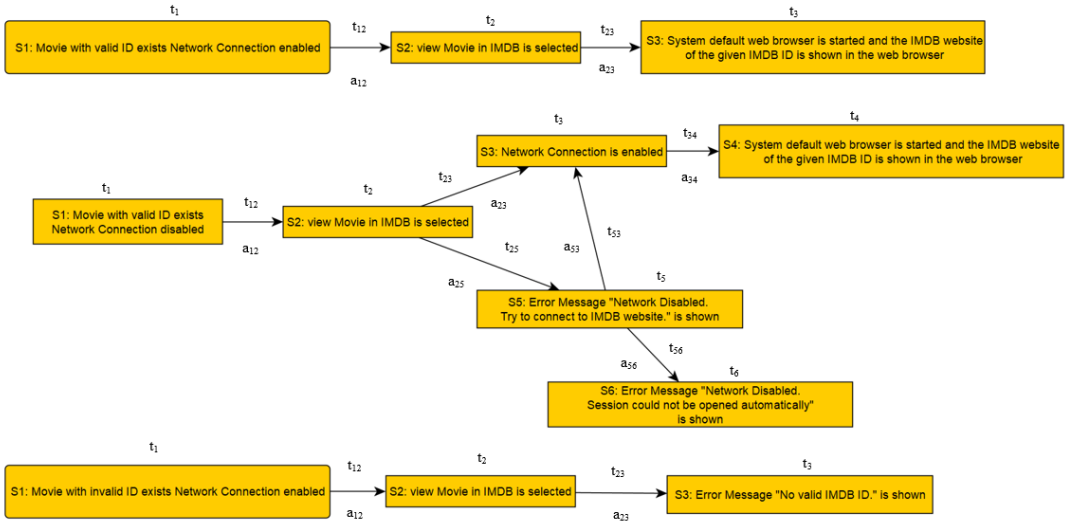

Figure 4.2: TUM for the systemfunction "view Movie in IMDB" in Movie Manager

The middle graph in Figure 4.2 shows the time dependency: As soon as t5 > 120 (unit in sec.), the state changes from S5 to S6. If the network connection has been re-established before 120 sec. have elapsed, the system switches to state S3. The graph includes all sequences that are conceivable when calling the system function "view Movie in IMDB". In the next step, a test suite could be automatically created from the graph using EXAM by traversing all paths in the graph. The exact procedure of this step is not described in detail in the approach.

4.4 Approach 2: A Model-Based Testing Technique for Component-Based Real-Time Embedded Systems

4.4.1 Description

Component-based modeling of embedded systems is becoming more and more important in the field of "software engineering" due to the increasing complexity of the systems. The approach now shows possibilities to improve the quality of tests of component-based embedded systems by means of several graph-based test models. The focus is particularly on Non-Functional requirements, more precisely, real-time requirements. The appropriate testing of these is often neglected during the integration of the individual components in a real-time embedded system. The test models presented here (except for the CIIG (Component Interface Interaction Graph), which only fulfills the former) take into account not only the functional and temporal dependencies between individual components, but also the temporal dependency between state transitions within a component. All the models presented are created manually using a tool. In the following, it will be briefly explained how and from which data the models are collected, which elements they consist of and in which respects they differ.

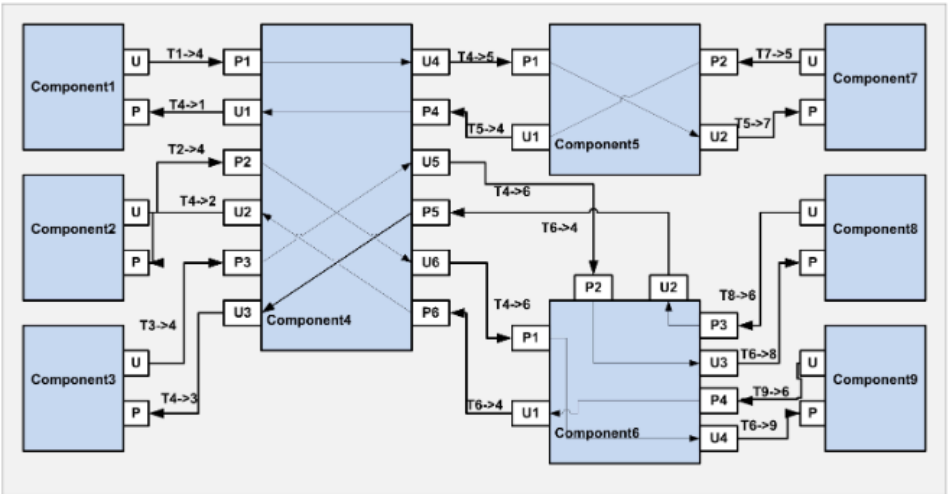

Figure 4.3: CIIG [30]

The CIIG, shown in Figure 4.3, can be derived from architecture framework and design specifications, that specify the primary components, interfaces and dependencies of softwaresystems, for example by context diagrams or sequence diagrams [30]. For the CIIG, a component subnet

is created for each component in the sequence diagram. Also, the external and internal message edges and provider/user interfaces can be derived from component level sequence diagrams. For each message in the sequence diagram, a component provider interface, a component user interface and an external message edge is created. Additionally, one internal component message edge is created between two external message edges.
The CIIG consists of a set of components, which in turn have a certain number of user (U) and provider interfaces (P). The paths in the model are defined via so-called message-edges. Internal message edges determine from which provider interface to which user interface the message can be sent in a component. External message edges determine from which user interface to which provider interface the message can be sent between individual components. Each external message edge is assigned to a time stamp (for example T1->4), which stores the transmission time of the message.
"A CIIG represents the time-dependent connectivity relationships between these components as well as time-dependent relationships inside a component [...]. But a CIIG does not reflect the internal behaviour of a component [...]" [30]. This is realized by a CSIBG (Component State-based Interaction Behavior Graph), shown in Figure 4.4.

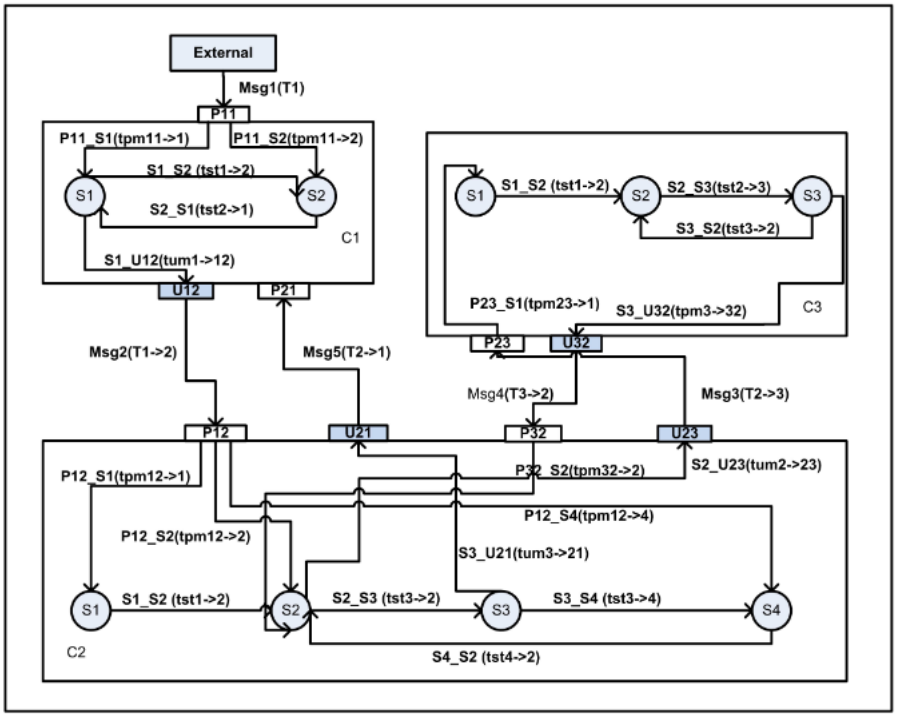

Figure 4.4: CSIBG [30]

"A CSIBG is created from a CIIG and state transitions in state diagrams for each corresponding component. Test paths are generated from the CREMTEG CSIBG diagram [...]" [30], since in this diagram the maximum number of possible paths that can occur through method calls or state transitions has been created: The number of test paths is equal to the product of the number of all possible transition paths in each individual CSIBG component subnet. In this model, in addition to the CIIG, all internal states (S), state transitions (for example S1_S2), transitions between Provider Interfaces and internal states (for example P12_S1), transitions between states and User Interfaces (for example S2_U23) time dependencies between individual

states (for example tst1->2) and between the states and the provider (for example tpm12->1) or user interfaces (for example tum2->23) of a component are represented. The most comprehensive model is the CSIEDBG (Component State-based Event-driven Interaction Behavior Graph).

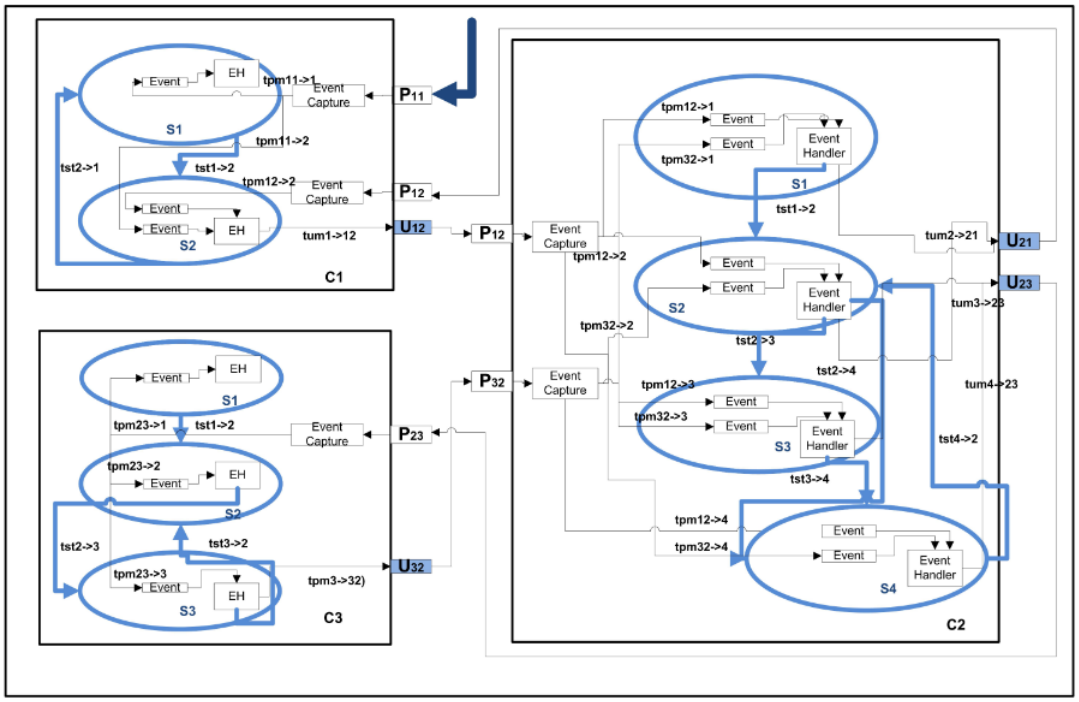

Figure 4.5: CSIEDBG [30]

"CIIG and CSIBG clearly described the time sequences of method calls and paths of state transitions, but they do not deal with concurrency" [30]. The CSIEDBG additionally considers multitasking and non-deterministic behavior, e.g., when processing simultaneously generated messages. Therefore, the model in Figure 4.5 is extended as follows: Each provider service in a component has one event capture. Each state in a component contains one event handler (EH) [30]. "The provider component processes the message by event capture to generate events in the current component state. All generated events are sent to event handler to decide whether to transition to the next state or send an outgoing message to the next component [...]. Decisions in the event handler are modeled as predicate expressions, where each clause is an event." [30]. New test paths are created according to CACC (Correlated Active Clause Coverage). CACC is a special form of predicate logic. For the exact definition of CACC, see the glossary.
The models are used to automatically generate test specifications for component integration testing [30]. The procedure is not described in more detail in the approach. The implementation of the test cases from the test specification is not automated. To evaluate the suitability of the models, the test suites are run under a system in which manual errors have been introduced [30]. Test Data were hand generated. "Test data were taken as inputs to a proprietary test automation tool [...]. The test tool reads the input test procedure files, launches the software, waits for the test to complete, retrieves output log files and analyzes the results to determine whether test was successful [...]. A final test report was generated after all tests were run" [30]. The results of the approach have shown that the function and block coverage and fault detection rate could be significantly improved by using a CSIEDBG compared to a CIIG or CSIBG.

However, in a complex real-time embedded system, it is often impossible to cover all test paths. For this reason, it must be decided on a case-by-case basis which test approach is chosen. Each model fulfills a test criterion of different quality: A CIIG fulfills the All-Interface Coverage Criterion (AIC). “This criterion ensures that each interface is tested once” [30]. A CSIBG fulfills the All-Interface-Transition Coverage Criterion (AITC). “This criterion ensures that each interface between components is tested once and each internal state transition path in a component is toured at least once” [30]. A CSIEDBG fulfills the All-Interface-Event Coverage Criterion (AIEC). “AIEC combines edge coverage with logical coverage” (see above) [30]. Models presented here serve as a basis for the creation of automatically generated test cases.

4.4.2 Application

The interaction between individual components is now applied to the Movie Manager Application, shown in Figure 4.6. In order to maintain clarity, the labels above the message edges are omitted as far as possible. The various widgets within the Movie Manager form the components in the overall system. Within these components there are states (e.g. under the component “Movies Tab” there is a state “empty” if no movies are stored in the movie list and a state “Movies List” if at least one movie is stored). The message edges show the information exchange of the existing system functions. Multi threading does not play a role in the Movie Manager, so a CSIBG was used for the demonstration.

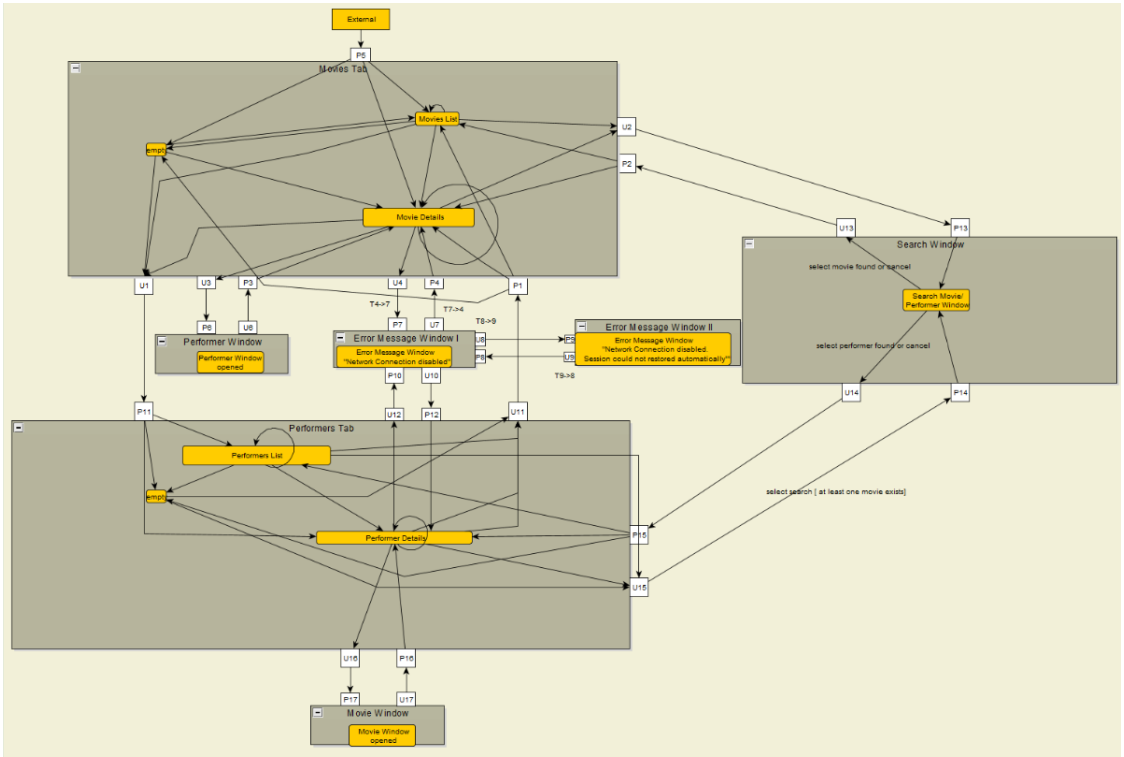

Figure 4.6: CSIBG for the Movie Manager application

Legend for graph shown in Figure 4.6: The grey rectangular boxes are components of the movie manager, the yellow boxes are the states in a component, the small rectangular boxes on the edge of a component are User- (U) or Providerinterfaces (P). Arrows symbolize intern (in a component) and extern (between components) messages. "T" are time stamps at external component messages.
The system functions "show Movie in IMDB" and "show Performer in IMDB", which have already been modified in approach 1, are extended by the following function: The error message window "Network Connection disabled" is to be closed automatically if the network connection could be restored before 120 seconds have elapsed. For this example (T7->4) - (T4->7) must be smaller or equal 120. The Message Edge between U8 and P9 only exists, if (T8->9) - (T4->7) is greater than 120.

4.5 Comparison

Table 4.2: Synthesis matrix

No.	Approach 1 (section 4.3)	Approach 2 (section 4.4)
1a)	Requirements specification: This could be natural language text, spreadsheets, drawings or UML charts. TUM: Will be manually derived from the requirements specification by a special tool. Test Cases: These are automatically derived from a TUM and executed by EXAM.	CREMTEGs were manually created with a tool from software architecture and design specifications (like Graphs, sequence diagrams, state diagrams, grammars, logical expresses and input domains). The CIIG can be derived from architecture framework and design specifications, that specify the primary components, interfaces and dependences of software systems, for example by context diagrams or sequence diagrams. A CSIBG is created from a CIIG and state transitions in state diagrams for each corresponding component. A CSIEDBG is an extended version of the CSIBG. It includes handling with events for multitasking. Test specifications were automatically developed from CREMTEGs. An test tool reads the input test procedure files, launches the software, and waits for the test to complete.
1b)	Requirements specification in different formats. System boundary is clearly defined.	System boundary must (particularly because the approach stays in relation with an embedded system) clearly be defined. Software architecture and design specification, from which the data for the models are derived.

1c)	The TUM model is created in two steps: First, the system boundary will be defined. Second, all sequences of stimuli and their responses across the system boundary will be enumerated. All stimuli that go across this boundary are extracted from the existing requirements. Each possible sequence of stimuli will be mapped to a response of the system. The sequences are created in ascending order of length. By this procedure a complete and consistent usage model is created. The finished TUM is passed to EXAM in the next step. After the import, test cases are automatically generated by running through the different paths of the TUM. For the test oracle, an appropriate counterpart from the EXAM test automation library is assigned to each individual stimulus. An importer allows thereby the import of a requirements library from a document-based requirements management system. The elements of this system can be traced back to each individual object in the TUM.	First the CREMTEGs will be constructed. For the CIIG, a component subnet is created for each component in the sequence diagram. A CSIBG is created from a CIIG and state transitions in state diagrams for each corresponding component.Test specifications for the test cases are automatically created from the finished CREMTEGs. The implementation of the test cases from the test specification is not automated. Finally, the test cases created from the CREMTEGs are run through in the SUT. Test Data were hand generated. "Test data were taken as inputs to a proprietary test automation tool [...] . The test tool reads the input test procedure files, launches the software, waits for the test to complete, retrieves output log files and analyzes the results to determine whether test was successful [...]. A final test report was generated after all tests were run"[30].
2a)	Automatically generated test cases are created from the requirements; testing possible at both system and component level. Test time reduced (through early error detection in requirements and the related incorrect creation of test cases). Avoidance of ambiguities in the description of requirements to prevent errors in test creation. Acceptance criteria such as test coverage can be achieved more easily with the help of the model. Requirements are less likely to be misunderstood during software development for the reasons mentioned.	The basis for the usage scenario is a component-based embedded real-time system. The test technique is used to put the internal processes (such as state transitions, event handling) in relation to the required time and thus create a more transparent test model, which improves the error detection rate in the system and the general code coverage of the system.
2b)	Test designer in the software team, Software developer, Requirements Engineer	Test design team (must be familiar with software architecture design e. g. to analyze UML diagrams, needs advanced domain knowledge)

2c)	Software Requirements → Requirements Analysis → Formal Analysis, Software Requirements → Requirements Validation → Model Validation, Software Testing → Test Levels → The Target of Tests, Software Testing → Test Levels → Objectives of Testing, Software Testing → Test Techniques → Model-Based Techniques, Software Engineering → Models and Methods → Analysis of Models → Analyzing for completeness, Software Engineering → Models and Methods → Analysis of Models → Analyzing for consistency, Software Engineering → Models and Methods → Analysis of Models → Analyzing for correctness, Software Engineering → Models and Methods → Analysis of Models → Traceability	Software Requirements → Requirements Analysis → Formal Analysis, Software Requirements → Requirements Validation → Model Validation, Software Requirements → Requirements Validation → Acceptance Tests, Software Testing → Test Levels → The Target of Tests, Software Testing → Test Levels → Objectives of Testing, Software Testing → Test Techniques → Model-Based Techniques
3a)	An editor for the creation of Timed Usage Models, Requirement Management System, EXAM, appropriate counterpart from the EXAM test automation library	Editor for creating the CREMTEGs, Program for generating automatically test specifications from CREMTEGs, tool which starts the test and determines whether test was successful
3b)	The creation of the TUMs is achieved using an editor, but must be generated manually. EXAM includes the automatic generation of platform dependent code and the automatic execution of the raised test suite completely without human interaction.	The CREMTEGs are created manually using an editor. The creation of the test specification from the CREMTEGs is automated. The implementation of the test cases from the test specification is not automated. Test value generation and fault insertion must be done by hand. A tool starts the test and determines whether test was successful automatically.
4a)	The approach was evaluated in the testing of a "start-stop functionality" (power train functionality) and energy management in a car. The degree of coverage of the requirements in manually created test cases was compared with the resulting degree of coverage in the automatically generated creation of test cases based on TUM. The advantages of a TUM were already evaluated during its creation from the requirements.	The approach was evaluated on an existing component-based embedded system (satellite communication system written in C and C++, 75000 lines of code). The system was defined using sequence and state diagrams. The system was then modified by mutation-based fault implementation to verify the suitability of the design-based test models.
4b)	The automatic creation of test cases covers a wider range of operation sequences. The coverage of requirements was systematically increased. System responses caused by the given functionality were partially undefined (discovered during the creation of TUM). Requirements were cleansed of deficits, shortcomings, contradictions and ambiguities.	The use of extended design-based models in the generation of test cases achieves a better error detection rate and a higher code coverage (both function and block coverage) than the manual collection of test cases. The interaction of internal states of the system (i.e. in the individual components) could be made more transparent.

Both approaches are presented to automate tests while taking a temporal component into account. Approach 1 places particular emphasis on analysing and "cleaning up" the existing requirements of ambiguities, deficits and shortcomings. The procedure in this approach is partly manual, partly automated: TUMs are created from the existing requirements using an editor. Design errors of the requirements can be detected during the creation of the model. EXAM allows the automatic import of the TUM. EXAM generates code for the implementation of the test cases and executes the test suite automatically. The level of automation of the approach is therefore very high. In the second approach, Component-Based Real Time Embedded Model-Based Test Graphs are generated manually from the existing architecture and design specification. Similar to the first approach, requirements are unified and thus freed from ambiguities. Real-time requirements, i.e. requirements that are linked to a time condition, are the main focus of testing. The level of detail of a CREMTEG CSIBG or CSIEDBG is significantly higher than that of a TUM. The special feature of the approach is that for the first time information from component level sequence diagrams and statecharts are combined to derive a graph based test model with timing properties for test generation. From the CREMTEGs, a test specification for component integration testing can be automatically generated. The test suite must still be created manually from the test specification. Test data and seeding of faults in the system are also created manually. The second approach therefore has a low level of automation. The use of advanced design-based models (supporting AITC and AIEC) in the generation of test cases achieves a better fault detection rate and a higher code coverage (both function and block coverage) than the method used AIC.

4.6 Conclusion

The literature search has shown that the topic of "testing with a timing component" is not yet particularly widespread in the industry. The search query "test AND requirements AND 'Timed Usage Model'" yielded only two search hits, whereby both articles found were by the same author. The second approach describes that it is very difficult to get access to running real-time embedded systems, which are necessary to evaluate the approach. Some intermediate steps have to be done by hand. Therefore, this approach does not seem to be very mature yet. The test design team also needs advanced software architecture skills to be able to analyse UML diagrams. The first approach is a step further in the automatic creation of executable tests: The TUM is given to EXAM, which executes all intermediate steps fully automatically until the test suite is executed. Unfortunately, the text only very sparsely or not at all discusses EXAM's procedure. However, in both approaches, the quality of the requirements could be improved considerably through the use of uniform models.

5 Testing with a classification tree

5.1 Introduction

If one wants to ensure that all possible cases are covered and checked in a test, the number of test cases can quickly become unmanageably large. If you have a certain number of parameters to consider in your tests, you have to try out and examine every possible combination of these tests to make absolutely sure that no unforeseen problem occurs. If parameters are also continuous, there is also the question of which intervals and values to consider. Depending on the resolution, this can be an unlimited number.

Classification trees (CT) offer an approach to reduce the complexity of such parameter constructions and to keep them clearly arranged. Test cases covering the relevant parameter constructions can then be derived from the CT. This chapter reviews the use of such CTs based on two selected articles describing their uses and suitability.

First, the literature search in the context of classification trees is explained. Subsequently, section 5.3 deals with classification trees as a method for creating test cases for model-based systems and discusses them. Using the movie manager as an example, the classification trees are applied to certain requirements. Chapter 5.4 then introduces Event Sequence Graphs (ESG) as an alternative concept for model-based systems. These are also applied to the movie manager example before a comparison of both approaches takes place in section 5.5. Finally, a conclusion is drawn in section 5.6. There, we conclude by answering the question of the extent to which CT helped reducing a complex problem to a manageable number of test cases. As with the previous chapters, an attempt is then made to apply the design of this approach to the Movie Manager example.

For a definition of the terms anti-lock braking system (ABS), classification tree (CT), event sequence graph (ESG) and model based black-box testing (MBBBT) please see the glossary.

5.2 Literature search

The literature search was conducted via snowballing on the given article “Systematic Model-Based Testing of Embedded Automotive Software” and a keyword search. Two partial approaches were used in snowballing. On the one hand, forward snowballing was used to find articles referencing the first given article. On the other hand, backward snowballing was used to search the articles that are listed by the initial article in the bibliography. Thus, publications directly related to the given article were considered. An overview of the main search results can be found in Figure 5.1.

Website	Search date	Search limit	Search term	# Hits	# Relevant hits	# New relevant hits	articles	Comment
	22.11.20		Backward Snowballing	14	2	2	Conrad, M., Dörr, H., Fey, I., Yap, A., "Model-based Generation and Structured Representation of Test Scenarios"	
							Grochtmann, M., Grimm, K., "Classification Trees for Partition Testing"	
	22.11.20		Forward Snowballing	25	1	1	Belli, F., Hollmann, A., Kleinselbeck, M. „A graph-model-based testing method compared with the classification tree method for test case generation"	
Elsevier	22.11.20		("classification-tree" OR "classification tree") AND ("model-based" OR "model based") AND "test design"	21	0	0		Interesting Articles behind Paywall
Elsevier	22.11.20	Datum: 2015 – 2020	("classification-tree" OR "classification tree") AND ("model-based" OR "model based")	972	0	0		Only first 50 hits examined
IEEE	22.11.20		("classification-tree" OR "classification tree") AND ("model-based" OR "model based")	68	1	0	Belli, F., Hollmann, A., Kleinselbeck, M. „A graph-model-based testing method compared with the classification tree method for test case generation"	
ACM	22.11.20		("classification-tree" OR "classification tree") AND ("model-based" OR "model based")	112	1	1	Thi Bich Ngoc D, „Constructing test cases for n-wise testing from tree-based test models"	

Figure 5.1: The tabular documentation of the search. The relevant results are recorded on the basis of the parameters specified in the columns.

A term based search was then performed on Elsevier, IEEE Xplore [34], and ACM digital library [3]. The search query reflects the exact wording that was finally used. During this process, experiments were conducted to estimate which keywords were relevant enough to elicit a suitable number of search hits. In contrary, backward snowballing proceeded without any problems by searching and evaluating the sources listed in the bibliography of the main article in order. Forward snowballing was performed by searching the citations given at Elsevier.

Figure 5.1 shows the relevant results of both search methods. The title and abstract of all results have been viewed, aside from the 1000 hits on Elsevier. There, the first fifty results have been considered. The assessment of whether an article is relevant was determined on the basis of three criteria:

1. CT must be used or described to allow conclusions to be drawn about their suitability

2. The article deals with testing or with the test design

3. The article must be available

The first criterion is intended to filter out articles that do not deal with CT at all and thus do not allow any conclusions to be drawn. The second criterion is to filter out those documents that do not deal thematically with test cases or even software development. Often these articles deal with other types of “Classification Trees”. An example here are decision trees or graphical visualizations of the classification of objects in artificial intelligence. These trees describe other phenomena or other problems. However, this chapter is explicitly about the use of such trees for the generation of test cases.

Both approaches to the search yielded multiple hits. One of the articles listed among the relevant search results occurred in the search using both approaches, giving it additional relevance to the topic. Accordingly, the article “A Graph-Model-Based Testing Method. Compared with the Classification Tree Method for Test Case Generation” by Belli and Hollmann[10] has been selected.

5.3 Approach 1: Systematic Model-Based Testing of embedded automotive software

5.3.1 Description

The first approach comes from Conrad et al[20]: The article "Systematic Model-Based Testing of Embedded Automotive Software" deals with the application of CT to problems that have a simulation or model of the application. The application example of the paper is the testing of an Antilock brake system (ABS). Therefore, the introduction starts with a comparison of traditional and model-based software development. Figure 5.2 from the article shows this difference well. The necessity of model-based test methods is thereby justified. The approach introduced as "model-based black-box testing" (MBBBT) is presented below. Two perspectives are presented: Requirements-based test design and model-based test design. The basis for this is the creation of a CT.

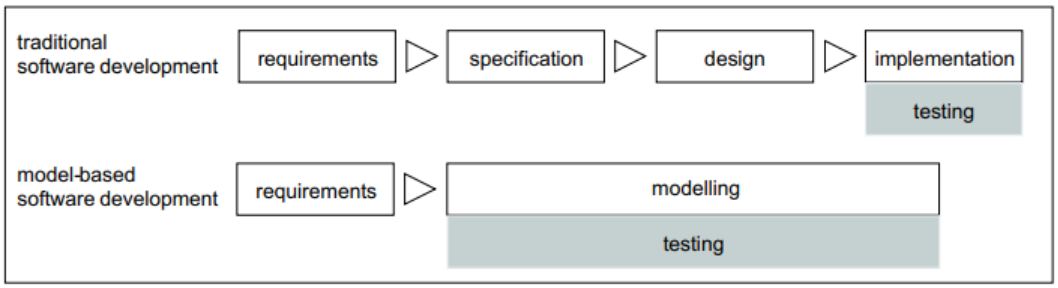

Figure 5.2: This figure from Conrad[20] shows the difference between traditional development and model based developement. They examine CT for projects with a model of the application. [20]

A graphically represented CT consists of the following components: The root denotes the product or model. The branches derived from it define the classifications. Classifications describe different requirements of the product. This depends on which of the two perspectives mentioned above was chosen as the basis. The leaves of a CT describe the different classes. A class in this context is a qualitative description. Thus, a category is described by the classes here. We can also say the range of values of a category is decomposed into the different classes. Figure 5.3 shows the requirements-based CT for the ABS model. At the top of the tree are the individual classifications with the different values for the classes. Because this is a requirements-based CT, mostly qualitative descriptions such as "dry", "straight", or "strong braking" are found as leaves.

For the test cases, the combination of these leaves is now to be tested. If you test all combinations of classes of this CT, you cover all test cases defined via the requirements. This assumes that the requirements are complete. In this example, this results in $4 * 2 * 3 * 5 = 120$ combinations.

The next step is the generation of a CT more suitable for the actual model that is used in the developement. Classification now describe different parameters. Classes contain quantitative value or an interval for the input parameters of the model.

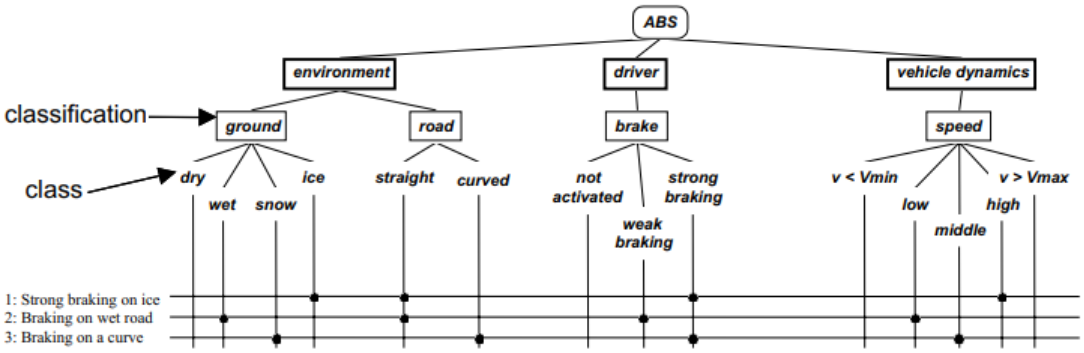

Figure 5.3: This figure shows the CT made for the use case as presented in [20]. The top part shows the different classifications and classes of the tree while the lower part shows the construction of test cases from the combination of leaf nodes.

Thus, for both perspectives, a CT is created and compared to analyze whether the requirements are covered by the model-based test scenarios. In this process, the classes of the requirements tree are mapped to the model tree. A model-based tree, on the other hand, describes the input dimensions or model parameters. Qualitative descriptions must therefore be transformed into the input parameters of the model. [20] discusses 1:1 and 1:n relations. 1:1 relations represent unique mappings, while 1:n reflects an occurrence of the corresponding requirement class in multiple parameters of the model. If this mapping is successful, a requirement class can be covered by testing the related classes in the model tree. The relations of the example classes of both the requirement and the model CT are summarized in a table. The summary of [20] explains the advantages of the MBBBT despite higher expenditure by the classification of two different approaches and recommends the application of this approach with the software development with models.

5.3.2 Application

The approach of a CT requires as a basis a model for frequently testing. After the approach of Conrad et al.[20], requirements, which can be mapped on it, are also needed. Test cases are then determined from this by combining the classes. Tests are conducted with the model. In this example, there is no real, simulatable model to describe it, but there is a description of requirements that could be mapped into a CT. The requirements described are:

1. Movie
    a) A movie must be able to be added including description.
    b) A movie must be able to be removed
2. Performer
    a) It must be possible to describe a performer
    b) A performer must be able to be linked to a movie
    c) An actor must be able to be removed

3. Previous movies

    a) Previous movies must be saved

4. Rating

    a) It must be possible to rate a movie

    b) It must be possible to display the rating of a movie

The CT method appears to be more suitable to single functions. The reason for that is, that the CT makes more sense if all leaf combinations are useful. This is more difficult to achieve if multiple independet functions are achieved. Therefore, the application of the CT approach focuses on the function "Add and describe performer" as the main functionality of the second requirement. The resulting CT can be found in Figure 5.4. Three questions are important for this functionality: Does the performer already exist? What movies are linked to them? How complete is their description? The function must be able to deal with these different aspects. Therefore, the tree contains these parameters.

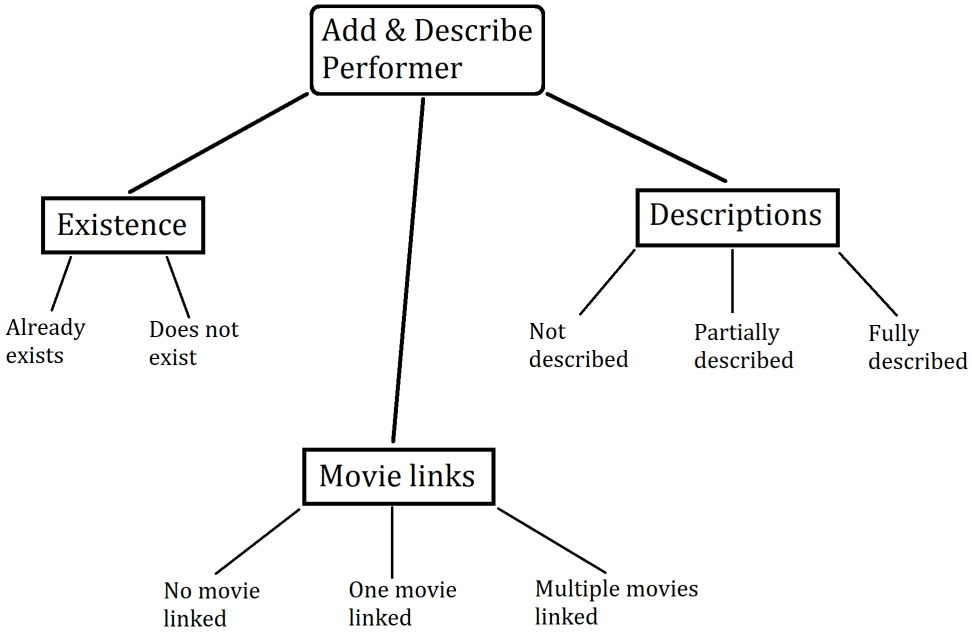

Figure 5.4: The requirements-based CT for the Movie Manager example defines the various classifications and classes for the test cases. Only a transformation into model parameters has to be done before the test cases can be constructed.

In the absence of a proper model against which to make classifications, exactly what counts as input or class depends on the exact implementation. A possible transition to a model based tree according to Conrad et al. [20] can be found in the following table:

The one 1:n relation means, that a partially constructed list can contain different combinations of list entries. These may play a role, so that the model actual needs to test more combinations of partially filled lists. Based on this transition, it is then possible to determine suitable test

Table 5.1: A tabular listing of the various requirements with example values. The last column indicates which type of transfer is possible with a suitable implementation.

Classification	**Class**	**Model values**	**Relation**
Existence	Already exists	Valid	1:1
	Does not exist	NULL	1:1
Descriptions	Empty	[""]	1:1
	Partial	["Name":"Max M."]	1:n
	Complete	["Name" : "Max M.", "release date" : "20XX", ...]	1:1
Links	No movie	NULL	1:1
	One movie	["X: The beginning"]	1:1
	Multiple movies	["X: The beginning", "X: The continue", ...]	1:1

cases. This results from the combinations of the different classes. Assuming that n is only 1 for the 1:n relation, the classifications for the existence of the performer, the linked movies and the description result in $2 * 3 * 3 = 18$ combinations of test cases. Individual test cases can then be, for example, the following:

- Adding a performer that already exists, without links to movies and a partially filled description, e.g. name and age only.
- Adding a new performer without details while trying to link him with a movie.

These are individual test cases. If you work through all combinations, the tests are considered complete. In some cases, its not necessary to test ALL combinations, since some combinations may not be possible or distinctive. It's heavily depended on the model or the requirements in general.

5.4 Approach 2: A Graph-Model-Based Testing Method compared with the Classification Tree Method for Test Case Generation

5.4.1 Description

The selected article "A Graph-Model-Based Testing Method Compared with the Classification Tree Method for Test Case Generation" starts with a description of the difficulties and complexity of testing under time pressure. Quickly, a newer method, "Event Sequence Graphs" (ESG), is introduced in addition to CT in the context of electronic control unit development. The next chapter explains some basics for this. Besides an explanation of CT, this section also contains a formal description of ESG. Here, a graph with its nodes represents events, including start and end events. Start nodes are the user inputs and end nodes are the final system responses.

The graph thus contains valid system flows. Taking any path from start to end event should result in a successful sequence of operations. For each graph a complementary graph can be constructed. The paper also discusses analyzing the complementary graph containing invalid system flows. The article describes valid tests as those that successfully traverse an ESG from start to finish or successfully cause the flows of the complementary graph to fail. Figure 5.5 shows a small example of how such a graph may look.

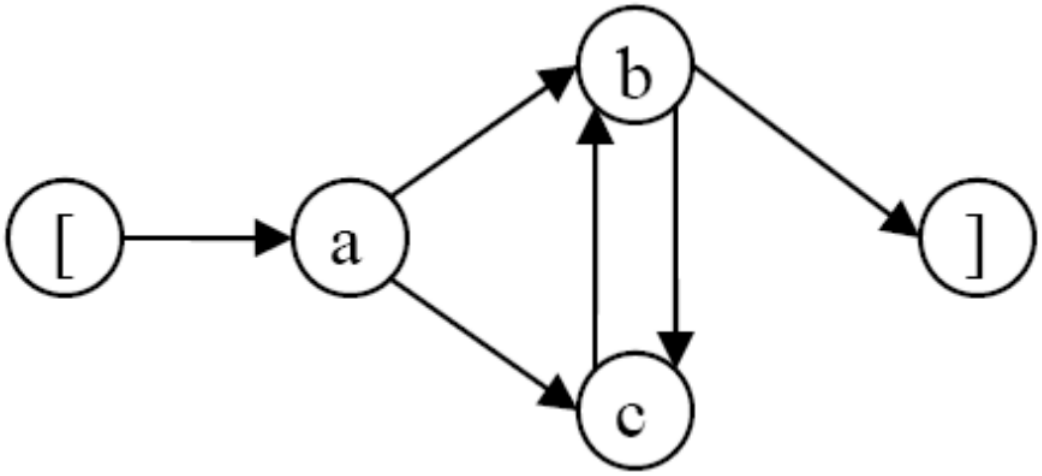

Figure 5.5: A simple ESG as an example from Belli and Hollmann. Nodes with brackets represent start and end events. A successfull test is performed by reaching the end node. [10]

While CTs look at different combinations of input values, ESGs look at combinations of different sequences of inputs. They then evaluate whether this sequence of "events" resulted in a success or failure of the action and whether that was part of the expected behavior. Unlike CT, which only looks at what should work, ESGs thus also consider failures.

Then, after the formal description, the test scenario is presented: A control system with ABS, ESP and an adaptive control element. This is followed by a description with specific characteristics of the test environment. The study starts with the application of ESG and CT to the test environment. Difficulties with CT are mentioned, citing dependencies that are difficult to understand. As a result of the study, a lower effort for the creation of CT test, but also a higher number of test cases and thus test steps are mentioned. Both approaches uncovered approximately the same errors. Figure 5.6 shows the main findings of the comparison from Belli and Hollmann's[10]'s test study.

	Manually test design (CTM)	ESG
Effort for the creation of the test design in hours	120	152
Number of test cases	26	11
Number of test steps	240	62

Figure 5.6: This table of Belli and Hollmann[10] shows the results of the comparison between CT and ESG.

The figure shows the result of the case study performed by Belli and Hollmann[10]. According to them, CT and ESG offer their advantages and disadvantages. The effort required for test design is slightly higher for ESGs, while ESGs perform better in terms of the number of test cases and test steps. This shows that CT can at least keep up with ESG in comparison.

5.4.2 Application

The CT, as they are built and used here, are identical in their function. In the article, a model-based CT is built directly without looking at the requirements themselves. This makes it all the more necessary to have a model for simulation or a ready-made application environment. Nevertheless, the same table as in Figure 5.5 would arise at this point.

Regarding the ESG, similar difficulties appear when trying to adapt this model based approach to the move manager example. Figure 5.7 shows an example for a partially constructed ESG. Test cases can be constructed by triggering events with Inputs or user actions. Test cases are successful, if they reach the exit event on the right side. Such a test could be:

- 1. Starting with an “empty data base”
- 2. Adding a movie, reaching “movie exists”
- 3. Adding a performer, reaching “Movie and performer exist”
- 4. Linking movie and performer
- 5. Closing the application without and error, leading to the exit event.

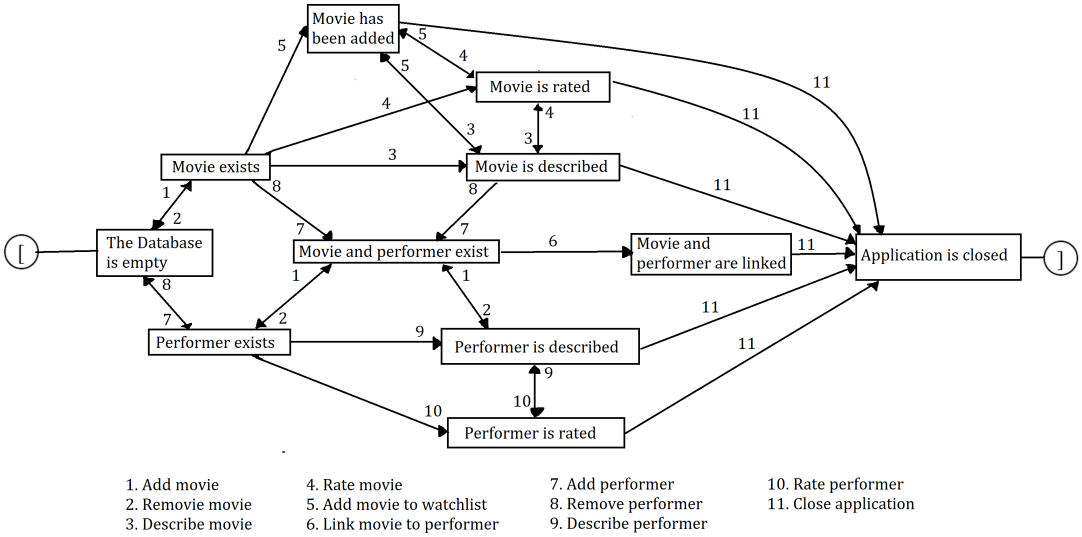

Figure 5.7: An example for possible event sequences to construct test cases.

5.5 Comparison

Table 5.2: The synthesis matrix summarizes the differences, similarities and peculiarities that stood out during the synthesis.

No.	Conrad and Fey	Belli et al.
1a)	Requierement-based CT, model-based CT, 1:1 and 1:n relations	CT, ESG, tabular comparison
1b)	Simulatable model exists, requirements can be mapped onto model	Simulatable model exists
1c)	1. Design of two CT for requirements and model. 2. Mapping of requirements to model. 3. Testdesign by combining all classes	1. Design of CT and ESG 2. Testdesign by combining all classes / state transitions 3. Comparison in a table
2a)	Testdesign with (black box)-models	Testdesign with (black box)-models
2b)	Softwaredeveloper, softwaretester	Softwaredeveloper, softwaretester
2c)	Software Testing, Software Maintenance and Software Construction	Software Testing and Software Maintenance
3a)	Matlab Simulink	CANoe
3b)	1. Manual , 2. Manual , 3. Automatable	1. Manual, 2. Fully automatable, 3. Manual
4a)	Comparison of both classification trees	Comparison of test statistics
4b)	Very suitable for black box models based on requirements	CT are a good alternative to ESG. CT are faster to develope, but result in more tests

Focus In [20], the special feature is the mapping from requirements-based CT to model-based CT. This is used to ensure that requirements are met by testing a model. From this, a procedure can be derived to formulate a CT from the requirements and apply it to the input parameters of its model or product. In the end, the use of CT for problems with models is recommended.

In [10], the main focus is the integration of ESG and its comparison with the application of a CT. With this, advantages and disadvantages with related methods can be determined.

Commonalities Both approaches deal with the application of CT to model-based test designs. As a result, the prerequisites are similar. The supported usage scenarios and stakeholders are also the same. This is mainly due to the fact that both approaches primarily target an application within the automotive industry. The level of automation is similar in that the test design seems harder to automate due to the level of abstraction.

Differences With its focus on the ESG, [10] clearly differentiates itself from [20]. As a result, the evaluation also focuses on comparison, while [20] focuses more on usability. Different tools and programs were also used to reach the goal. [20] also goes more intensively into the concept of CT. This is recognizable by the fact that the determination of a CT is dealt with in more detail via a transfer of the requirements.

ESG itself is distinguished from CT, as described in the previous sections, by considering a combination of possible state and input sequences instead of a combination of possible input values, which also makes it possible to ensure that cases that should not work do not work.

Table 5.2 shows the synthesis matrix of the two articles on which the comparison of this chapter is based.

5.6 Conclusion

The search on snowballing is promising and also offered relatively many relevant articles. With a term based search, the risk of being flooded with irrelevant hits is greater. This makes sense in that snowballing exploits a direct link via the citations, and free search does not require articles to be related. Thus, articles about decision trees or about testing thematically inappropriate use cases such as medical procedures also appeared among the results. In contrast, one has a wider reach, provided one chooses the search terms appropriately.

Classification Trees offer the possibility to reduce the number of test cases. It turns out that CT are well suited for model-based problems, but it can be difficult to represent arbitrary tasks in this way. [20] shows for this case how to transform requirements into a CT for the input parameters of a system model. However, the existence of a model-like description remains necessary. According to [10], it is shown that CTs nevertheless remain relevant, even alongside more recent methods. Comparing ESG with CT, Ct appear more suited for a fast classification of input parameters. The approach to derive a CT from given requirements makes it easy to derive test cases. Belli and Hollmans approach with ESG are more complicated to generate. It's harder to make sure to have a complete representation of all states and events as well. The amount of test cases also rises exponentially, since it's a combination of leaf nodes. Therefore, for smaller projects CT seems to be more suitable, while ESG can be helpful for bigger projects.

6 Testing with a formal specification

7 Testing with System Models

In this chapter, we aim to study the topic *testing with system models*. Section 7.1 presents the introduction of this topic. Section 7.2 describes how the systematic literature research is carried out to identify the existing approaches for testing with system models. In section 7.3, the first relevant approach for this topic, which is given from the Chair of Software Engineering of University Heidelberg to be studied, is described and implemented on the example application *Movie Manager*. In section 7.4, the second relevant approach, which is identified through the literature research, is studied likewise. Then, section 7.5 compares the two relevant approaches selected for this topic. In conclusion, section 7.6 presents the insights gained from this study.

In order to get familiar with *system model (SM), test model (TM), model-based testing (MBT), Unified Modeling Language (UML), Systems Modeling Language (SysML)* and *Object Constraint Language (OCL)*, please visit the glossary.

7.1 Introduction

The key to successful product engineering in the software industry today is in many cases a good quality-assurance and deployment of software systems [2]. In order to ensure the quality of a product and verify that the product meets its requirements, we need software testing.

Model-based testing (MBT) is a software testing technique that has gained much interest in recent years by providing the degree of automation needed for shortening the time required for testing [2]. It provides automatic generation of tests from models representing the behavior of the system under test (SUT) [1]. From these models, test cases can be derived directly or via some model transformations following different coverage criteria.

As our topic is testing with system models, it is important to differentiate the two model types *system model* (SM) and *test model* (TM) with respect to the MBT. In our first approach [2], which is presented in section 7.3, the authors explicitly stated that they used SMs instead of TMs. And the difference between SM and TM is explained as follows: "The difference between the two being that the former is both used for development and testing, whereas the latter is only used for testing" [2].

In order to discuss the differences between using SM and TM in particularly, the authors of [2] provide with some additional authors the article *Model-Based Testing using System vs. Test Models – What is the Difference?* [43], which is newer than [2] and presents two model-based testing case study examples which use SMs and TMs, respectively. In [43], the SM case study example is our first approach [2] to be studied within this chapter.

The article [43] summarizes the differences between the SMs and TMs as shown in Figure 7.1.

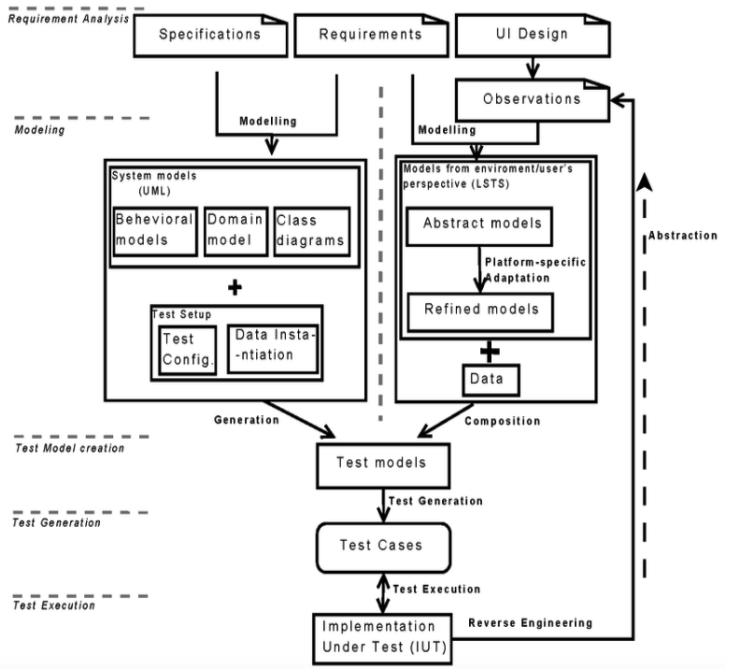

Figure 7.1: Sytem models and test models

From the viewpoint of modeling, one difference between SM and TM is in the way the expected behavior of the SUT is specified with respect to its interfaces; SM provides an internal viewpoint, whereas the TM provides an external viewpoint of the SUT [43]. In the terms of reactive systems, TMs provide stimuli and observe the SUT reactions, while the SMs expect the stimuli and provide reactions. Concerning the purpose of modeling, the TMs are developed solely for testing while SMs can be primarily developed for system development (however, SMs are simpler and more abstract than implementation models) and then used for testing as well [43]. However, there also some definitions in [43], which conflict with the usage and the way of the models are created in [2]. According to [43], in TM based approaches, implementation is seen as a black-box thus it is hard to give any verdict about how much of the implementation code has been covered by generated test cases unless source code is instrumented for this purpose. Therefore, requirement coverage is mostly used in this case. On the other hand, in SM based approaches, both code and requirement coverage can be observed, again provided that the implementation code is available for such analysis.
But as can be seen in section 7.3, the authors are entirely concerned with the requirement coverage in [2] while using SMs. After stating this, we would like to clarify that for our study we stick to the definition and usage of SM provided in [2], as it is also suitable with the Figure 7.1.

As the SM of the SUT is typically derived from the informal requirements, it is important to trace how different requirements reflect in the models, on different perspectives and on different abstraction levels, and how the generated test cases cover different requirements [2], [43]. For a model-based testing perspective, traceability of requirements means that the generated test cases from the model are linked with the requirements [12]. Thus, traceability of requirements helps to achieve the right level of coverage and shows what requirement has been covered by what test [2]. It also enables us to identify which requirements have been successfully tested and which have resulted in failures [2]. Hereby, traceability of requirements is a pivotal aspect of MBT that allows one to ensure that all requirements have been tested [1].

7.2 Literature search

This section describes the systematic literature research carried out according to the *Guidelines for Literature Research of the Chair of Software Engineering* [17]. It aims to identify and analyze the scientific articles regarding testing with system models.

Research question and research strategy

The main goal of this search is to find relevant approaches for testing with system models, which are similar to the approach provided in the given article *Tracing Requirements in a Model-Based Testing Approach* [2].

Based on [2], which is our start paper for chapter 7 in this study, we derive the following research question presented in Table 7.1.

Table 7.1: Research question and search terms

Research Question	Which approaches exist to generate test cases using system models for the traceability of requirements?
Search Terms	system model* AND test* AND trac* AND requirement?

In order to answer the research question and find relevant articles, we use "keyword-based search" and "snowballing" methods. For the keyword-based search method, we determined the search terms as presented in Table 7.1. These search terms were derived from the research question to focus on a certain aspect of the given article [2] and to find a similar approach.

For the literature research, the digital libraries of the IEEE Xplore (Institute of Electrical and Electronics Engineers) [34] and ACM (Association for Computing Machinery) [3] were used as main sources, since they include the most extensive range of examined scientific publications and are widely recognized.

In this literature research, keyword-based search method and snowballing search method are conducted to find relevant articles in aforesaid libraries. With keyword-based search method, we search for the search terms presented in Table 7.1 with different combinations, since each digital library has a different search method, e.g. advanced search, or command search.

Snowballing search method is applied to the given article [2] to identify further relevant articles. With backward snowballing method, we go through the reference list of [2] and select the articles as relevant, which fulfill the relevance criteria determined during the literature research. The relevance criteria are presented in Table 7.2. With forward snowballing method, we go through the articles, which cite the given article, and again select the articles as relevant, if they fulfill the relevance criteria.

In order to ensure that the research methodology and its implementation remain consistent and comprehensible, we formulated four relevance criteria as shown in Table 7.2. An article is relevant, if it fulfills all criteria. The first two criteria are formulated to check the availability and the subject area of the articles. Since this work is to be written in English, the article to be used should be available in English. To reduce the comprehensive subject area to the relevant area for our topic, the article should have a clear focus on the area of software engineering as described in the second criterion. The third criterion aims to focus on recent work and to limit

Table 7.2: Criteria for selecting an article as relevant

No.	Description
1	The article is available in English
2	The article has a clear focus on software engineering
3	The article was published in the last fifteen years (2005-2020)
4	The article contains an approach for the traceability of requirements using system models to generate test cases, i.e. the article answers the research question

the search results. The last criterion is formulated to answer the research question and to find relevant articles, which particularly provide similar approaches to the approach presented in [2].

Research execution and literature results

First, the search was carried out using the keyword-based search method with aforesaid search terms. In Table 7.3, the results of this search are documented.
First column shows the search location, second column shows the date on which the search was carried out, and third column shows the applied restriction of search. Fourth column presents the exact search query of the search terms and thus shows how they were searched in the specified parts of articles. Fifth column shows the number of overall results, and sixth column shows the number of relevant articles. Finally, last column presents the number of articles chosen to be used in this study.

Table 7.3: Results of the keyword-based search method

Search Location	Search Date	Restriction	Search Terms	# Results	# Relevant Articles	Used Results
IEEE	20.11.20	2005-2020, Advanced Search	((((“Full Text Only”:“system model*”) AND “Document Title”:test*) AND “Abstract”:trac*) AND “Full Text Only”:requirement?)	64	2	0
ACM	20.11.20	2005-2020, Advanced Search	[Full Text: “system model*”] AND [Publication Title: test*] AND [Abstract: trac*] AND [Full Text: requirement?] AND [Publication Date: (01/01/2005 TO 12/31/2020)]	9	0	0
ACM	21.11.20	2005-2020, Advanced Search	[Abstract: system model*] AND [Publication Title: test*] AND [Abstract: trac*] AND [Abstract: requirement?] AND [Publication Date: (01/01/2005 TO 12/31/2020)]	46	2	0
IEEE	21.11.20	2005-2020, Advanced Search	((((“Abstract”:system model*) AND “Abstract”:requirement?) AND “Abstract”:trac*) AND “Document Title”:test*)	59	0	0

As shown in Table 7.3, in all of our searches we searched the search term "test*" in the *documentation title* part, and the search term "trac*" in the *abstract* part of the digital libraries, in order to find the closest approaches as possible to the presented approach in [2]. Although the "system model" is the most important search term for this topic, it was not written even in the abstract of the given article [2]. Therefore, with considering that, we decided to search this term also in the *full text* part, instead of only in the *abstract* part of the digital libraries. Consequently, the search terms "system model" and "requirement" were searched in the first searches only in *full text* part, then in the second searches in *abstract* part of the digital libraries.

After identifying relevant articles by using keyword-based search method, snowballing method was conducted to the [2], which is our start paper for this study, in order to find further relevant articles. The results of the search using snowballing method are detailed documented in Table 7.4.

Table 7.4: Results of the snowballing search method

Search Date	**Reference**	**Direction**	**Number of Citations**	**# Relevant Articles**	**Used Results**
19.11.20	[2] Tracing Requirements in a Model-Based Testing Approach	Backward	15	1	1
		Forward	11	1	0

Thus, after conducting the keyword-based and snowballing search methods, we could identify six relevant articles presented in Table 7.5.

Table 7.5: Results of the literature search

ID	**Title**	**Authors**	**Year**	**Source**
[2]	Tracing Requirements in a Model-Based Testing Approach	F. Abbors, D. Truscan and J. Lilius,	2009	IEEE
[1]	MATERA - An Integrated Framework for Model-Based Testing	F. Abbors, D. Truscan and A. Baecklund	2010	IEEE
[14]	A Subset of Precise UML for Model-Based Testing	F. Bouquet, C. Grandpierre, B. Legeard, F. Peureux, N. Vacelet, and M. Utting	2007	ACM
[15]	Requirements Traceability in Automated Test Generation - Application to Smart Card Software Validation	F. Bouquet, E. Jaffuel, B. Legeard, F. Peureux, and M. Utting	2005	ACM
[43]	Model-Based Testing using System vs. Test Models – What is the Difference?	F. Abbors, D. Truscan, J. Lilius, M. Katara, H. Virtanen, A. Jaeaeskelaeinen, and Q. A. Malik	2010	IEEE
[12]	Requirements Traceability in the Model-Based Testing Process	Eddy Bernard and Bruno Legeard	2007	DBLP[1]

The articles [2], [1], [14] and [15] were identified through the keyword-based search method, and [43] and [12] were identified through the snowballing search method. The first identified article [2] through the keyword-based search method was already given us for this study as start paper. Since [1] and [43] are from the same author group of [2], we didn't choose them for our study as a second approach to examine.

[1]dblp computer science bibliography https://dblp.uni-trier.de/

The articles [14] and [15] have also common authors, and one of them is the same author of the article [12]. Therefore, we excluded the [14] and [15], and chose only [12], *Requirements Traceability in the Model-Based Testing Process*, which is closest to the provided approach in the given article [2], and very suitable for our topic *testing with system models*.

Consequently, ***Tracing Requirements in a Model-Based Testing Approach*** [2] and ***Requirements Traceability in the Model-Based Testing Process*** [12] are the chosen articles to be studied in this chapter 7.

7.3 Approach 1: Tracing Requirements in a Model-Based Testing Approach

This section describes the approach presented in *Tracing Requirements in a Model-Based Testing Approach* [2], and its application on the *Movie Manager*, which is an application used by private individuals at home to manage their own film collection [18]. What this management includes is described as sub-tasks in the user task sheet of Movie Manager in [18].

7.3.1 Description

This approach was presented in 2009 by Fredrik Abbors, Dragos Truscan, and Johan Lilius in [2] for tracing product requirements across a model-based testing process. With this approach, the authors show how the informal requirements of the SUT evolve and are traced to system specifications and from system specification to tests during the test generation process. They also show how the test results are analyzed and traced back the specification of the system.

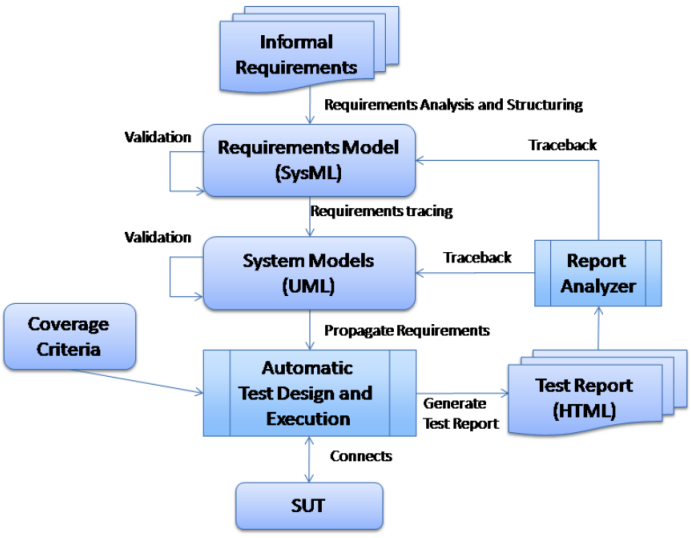

Figure 7.2: Overview of the model-based testing process [2]

As shown in Figure 7.2, the model-based testing process starts with the analysis and structuring of the informal requirements (including protocol specifications, standards, user scenarios, etc.) into a *requirements model* via Systems Modeling Language (SysML). Because compared to the

pure textual description of requirements, a requirement diagram offers several advantages due to its visual overview, e.g. missing aspects can be easily identified to help identify additional requirements, and the different types of relationships facilitate traceability and promote understanding [58]. Requirements traceability is built on top of this testing process, since authors want to be able to trace how different parts of the system models relate to the requirements and then to see how different requirements are covered by the generated test cases. Another reason for tracing requirements is that if a requirement changes, it is essential to know how this change is reflected in the models. Therefore, after requirements are structured hierarchically using SysML requirements diagrams, they are traced to different models or parts of the models implementing them.

For the modeling, the Unified Modeling Language (UML) is used to specify the SUT. For a successful test derivation, several perspectives of the SUT are modeled; a class diagram is used to specify a *domain model*, a *behavioral model* is used to describe the behavior of the SUT, *data models* are used to describe the message types exchanged between different domain entities, and *domain configuration models* are used to represent specific test configurations using object diagrams. In order to increase the quality of the resulting models, a set of modeling guidelines and validation rules are defined. These rules ensure that the models are consistent with each other and moreover, that they contain the information needed in the later phases of the testing process. For editing the SysML and UML models and for running these validation rules, the NoMagic's *MagicDraw*[2] tool has been used.

When all requirements have been linked to model elements and the models have been validated, the models used to specify the SUT are subsequently transformed into input for an automated test derivation tool for model driven testing, *Conformiq Qtronic*[3], via an automated transformation. Qtronic accepts as input a SM of the SUT from which it automatically designs test cases according to the selected coverage criteria. The input model can be expressed as a combination of UML state machines [43] and the transformation basically translates these UML models to the Qtronic Modeling Language (QML), a textual specification language with a Java-like syntax used by Qtronic for specifying the SUT. There are two main purposes for modeling behavior using state machines. First, by using UML state machines the behavioral properties of SUT specification are formally verified. Second, Qtronic tool expects the behavior of SUT in the form of state machines [44]. During the transformation from UML to QML, links between requirements and model elements are preserved.

In this approach, only the online testing mode of this Qtronic is used, in which tests are generated and applied on-the-fly against the SUT. For test generation, different coverage criteria types can be manually selected from the graphical user interface (GUI) of Qtronic like requirements, state, transitions, paths, conditional, or statement coverage. The generated test cases are sequences of input/output messages and their data values derived from the SMs to be sent/received by the SUT [43]. As the designed test cases are at the same abstraction level as the SM, an adapter is used to concretize the tests [43]. One-by-one Qtronic generates an input message, sends it via the adapter to the SUT, and generates a new input message based on the responses from the SUT. A logging back-end can be used during test execution. The logging back-end provides connectivity to the Qtronic reporting infrastructure and it is used by Qtronic to generate a test report. Three logging back-ends are provided by default. With these logging back-ends, Qtronic can generate test reports in HTML, SQLite, and XML format. When all tests have been applied against the SUT, Qtronic generates a test report in the chosen format, which summarizes the results of the testing process in terms of generated test cases, verdicts, coverage levels, requirements traceability matrix, etc. Unfortunately, the rest of the automatic test generation and execution process of this approach was not provided by the authors.

[2]NoMagic MagicDraw https://www.nomagic.com/products/magicdraw
[3]Conformiq Qtronic https://www.conformiq.com/

With this approach, it is also possible to trace-back requirements from test cases to models. For this purpose, the rest report is analyzed and the information of the failed test cases is collected. Then, the requirements attached to those test cases are traced back to system models. This enables to identify which requirements were not validated during testing and to what parts of the specification they are linked. A Python script was developed by the authors that automatically analyzes the Qtronic test report.

According to the authors, the provided approach proved beneficial through the fact that many errors have been detected in the early stages of the process, when the system models have been created. For instance [43], some errors originated from inconsistencies discovered in the SM, which were due to misunderstanding of requirements or to incomplete validation of models before testing. Other errors have been found in the adapter, as well as in the SUT.

Furthermore, it should be noted that although Conformiq is still active and offering different kind of products for an end-to-end test automation, its Qtronic product mentioned in this approach is not existing in Conformiq's current product list. Unfortunately, after our research, we could not find any information about what happened to Qtronic. Therefore, we assume that it was not developed further.

7.3.2 Application

First, functional requirements of the SUT should be analyzed and structured into requirements models using the requirements diagrams of SysML. Each requirement element contains a *name* field which specifies the name of the requirement, an *id* field that simply specifies the id of the requirement, a *text* field which describes the requirement, and a *source* field which specifies the origins of the requirement. The source can be a link to or a name of a textual document from where the requirement has been extracted.

In our implementation, we converted the user sub-tasks of Movie Manager into system functions in order to build a requirements diagram from system functional requirements. Figure 7.3 presents the example of a SysML requirements diagram for our application.

Then, the UML models of the SUT are built starting from the requirements models. During this process, the requirements should be traced to different parts of the models to point how each requirement is addressed by the models. In the provided approach, a set of modeling guidelines and validation rules should be defined for ensuring the quality of the resulting models. For this purpose, Object Constraint Language (OCL) is used. The aforementioned NoMagic's MagicDraw tool has been used for editing the SysML and UML models and for running the validation rules. But in our implementation of the approach, we use the free online tool of the diagram software *draw.io*[4] via *diagrams.net*[5] for this purpose.

[4]draw.io https://drawio-app.com/
[5]diagrams.net https://www.diagrams.net/

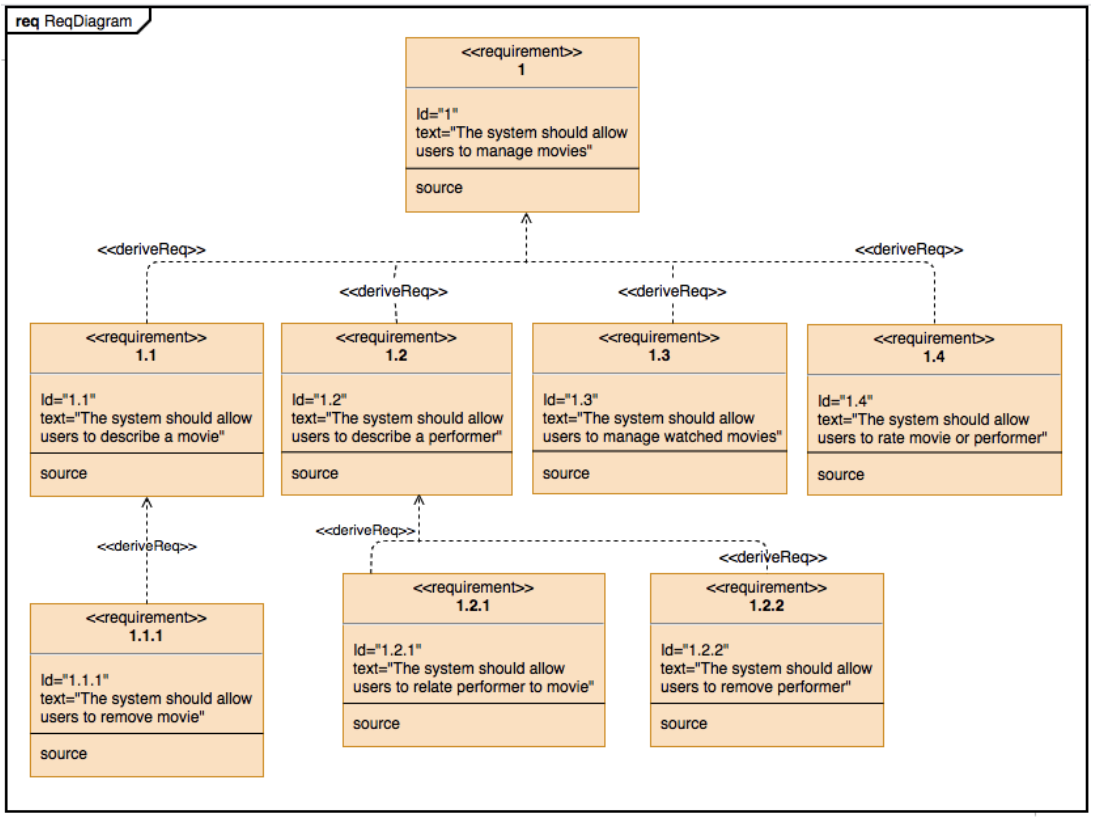

Figure 7.3: Example of a SysML requirements diagram

As the next step, a state machine model should be modeled from the requirements model, which is to be used as input for Qtronic, since the Qtronic tool expects the behavior of SUT in the form of state machines in order to be able to convert it into QML. In Figure 7.4, we present an example for the requirement with the requirement id "1.4" from the requirements model presented in Figure 7.3.

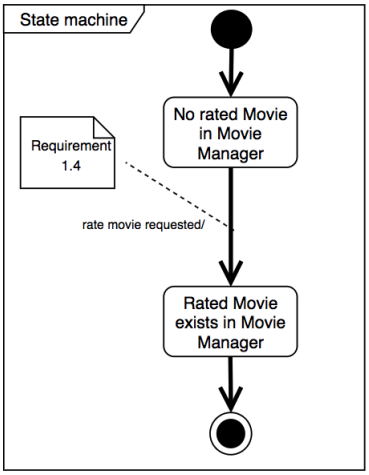

Figure 7.4: Example of a UML state machine

But as already mentioned in the description of this approach in subsection 7.3.1, Qtronic is no longer available and therefore we can not apply the remaining steps, which are *automatic test design and execution* and *generation of test report*. As we could not find any information on how Qtronic generates tests, we can not provide any exemplarily test cases either.

7.4 Approach 2: Requirements Traceability in the Model-Based Testing Process

This section describes the approach presented in *Requirements Traceability in the Model-Based Testing Process* [12], and its application on Movie Manager.

7.4.1 Description

This approach was presented in 2007 by Eddy Bernard and Bruno Legeard in [12] to automatically produce the traceability matrix from requirements to test cases, as part of the test generation process. Automatically generating the traceability matrix from requirements to test cases implies managing the links between the requirements specification, the model and the test cases. This approach focuses on this problem and is embedded in the *LEIRIOS Test Designer* technology, which is named currently *Smartesting*[6] [1]. The approach tags the dynamic part of the UML model with the requirement identifiers and uses it to produce automatically both the test cases and the traceability matrix.

First, the validation engineer constructs the model from a textual specification. The goal of this step is to translate the description of the features of the tested system into a precise UML model of the expected behavior [11]. The LEIRIOS Test Designer approach uses a subset of the UML 2.0 language with class diagrams, instance diagrams, state machine diagrams, and OCL expressions. While the OCL expressions within the class diagram formally describe the expected behavior of operations of a class using preconditions and postconditions, the OCL expressions within the state machine formalize guards and effects of transitions between the states.

Then, requirements traceability is managed by tagging manually the postconditions of the operations and the effects of the transitions in the UML model with the requirements. The format uses ad-hoc comment symbols to associate a requirement with an OCL statement which will be involved into a test target during the generation of the test cases. Once the model is reasonably trustworthy, the generation of tests is steered by the validation engineer on the basis of test selection criteria. Test selection criteria are supported by the LEIRIOS Test Designer tool to control the choice of tests from all the possible tests that can be derived from the behavior UML model [11], e.g. transition-based coverage, decision-based coverage, and data-oriented coverage. It is important to notice that all these test selection criteria are related to models and define how well the generated test suite covers the model [11].

The test generation method is provided by the LEIRIOS Test Designer tool. It consists in testing all the possible behaviors of the specification operations, by traversing the states of the system. This strategy is controlled by the previously defined test selection criteria.

[6] Smartesting https://www.smartesting.com/

This method is performed as follows:

- Partitioning of the model operation to generate all the possible expected behaviors,
- Computation of variable domain boundaries from each behavior (called test targets),
- Generation of test cases obtained, for each test targets, by traversing the underlying reachability graph of the model from the initial state to reach a state satisfying a test target.

Operations are the internal actions that modify the system state during the computation of user actions. They are called from the action part of transitions in the state machine and their meaning is defined by OCL postconditions in the class diagram [11].

A test case reaches a target, which involves OCL statements tagged with one or several requirement identifiers. Then, LEIRIOS Test Designer makes it possible to match a test with requirements during the generation of the test cases. In LEIRIOS Test Designer, a general framework to convert the generated test cases into executable scripts is also provided. The test script pattern is a source code file in the target language with some tags indicating where sequences of operation invocations have to be inserted. A traceability matrix is obtained after the tests are executed. For each requirement, the matrix gives the list of needed test cases to test it.

In [12], this presented approach is implemented on a simplified version of a drink vending machine controller, which is SUT, with "Buy a drink" scenario to illustrate the test generation process with LEIRIOS Test Designer. After expected functional requirements of the SUT are listed, it is modeled via a *class diagram*, an *instance diagram* and a *state machine*. Expected behaviors are specified by transitions (either external or internal) and those transitions are triggered by a *user event*, a *guard* and an *effect*. Requirements traceability is managed by tagging the effect of each transition with requirement identifiers. This link between the model and the requirements is used by the LEIRIOS Test Designer tool to produce the traceability matrix between generated test cases and requirements. At the end, from the SUT model, 12 test targets are generated, one for each transition, and 12 test cases are generated.

Unfortunately, the approach lacks the detailed explanation on how the test cases are generated and converted to executable test scripts via LEIRIOS Test Designer tool.

7.4.2 Application

In this part, we aim to to follow the same steps of the approach on our Movie Manager example as similar as possible.

First, we get the informal requirements of the system under test. To do that, we create use case diagram of Movie Manager using draw.io tool as shown in Figure 7.5. Our use case scenario: "User manages movies and corresponding performer data of a movie collection".

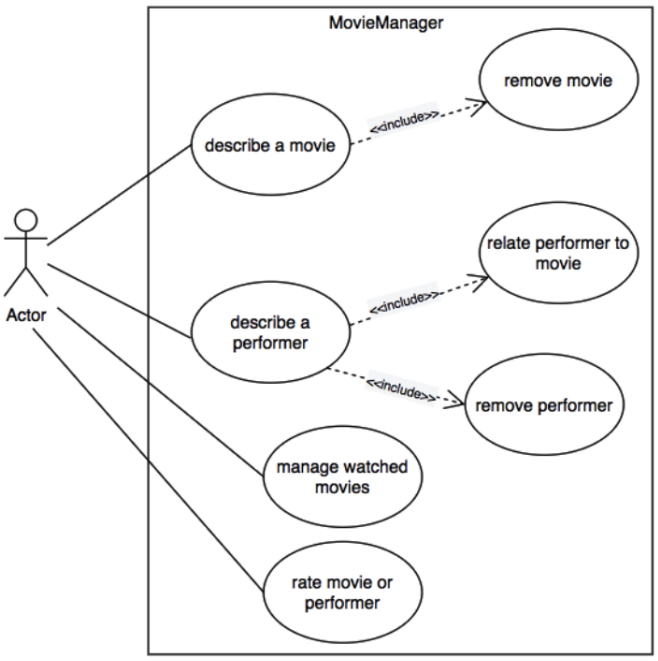

Figure 7.5: Example for a use case diagram of Movie Manager

The system under test is the Movie Manager. To precisely define the expected functional requirements of the Movie Manager, a list of requirements including Req Identifier, Req Name, and Req description is defined as presented in Table 7.6. For the requirements MM-2 and MM-7 we added some preconditions based on the Movie Manager [18] to be able to apply the next steps of the approach within this section.

Table 7.6: Movie Manager requirements

Req. Id	**Req. Name**	**Req. Description**
MM-1	Describe_Movie	The Movie Manager shall allow to describe a movie
MM-2	Remove_Movie	The Movie Manager shall allow to remove movie. If the movie is linked with a performer, the user is warned by Movie Manager. The movie is removed from the database after user confirms.
MM-3	Describe_Performer	The Movie Manager shall allow to describe a performer
MM-4	Relate_Performer_to_Movie	The Movie Manager shall allow to relate performer to movie
MM-5	Remove_Performer	The Movie Manager shall allow to remove performer
MM-6	Manage_Watched_Movies	The Movie Manager shall allow to manage watched movies
MM-7	Rate_Movie_or_Performer	The Movie Manager shall allow to rate movie or performer. If the movie or performer does not exist, the user is warned by Movie Manager

Then as an instance, we create the Movie Manager Class Diagram for the aforementioned requirements as presented in Figure 7.6.

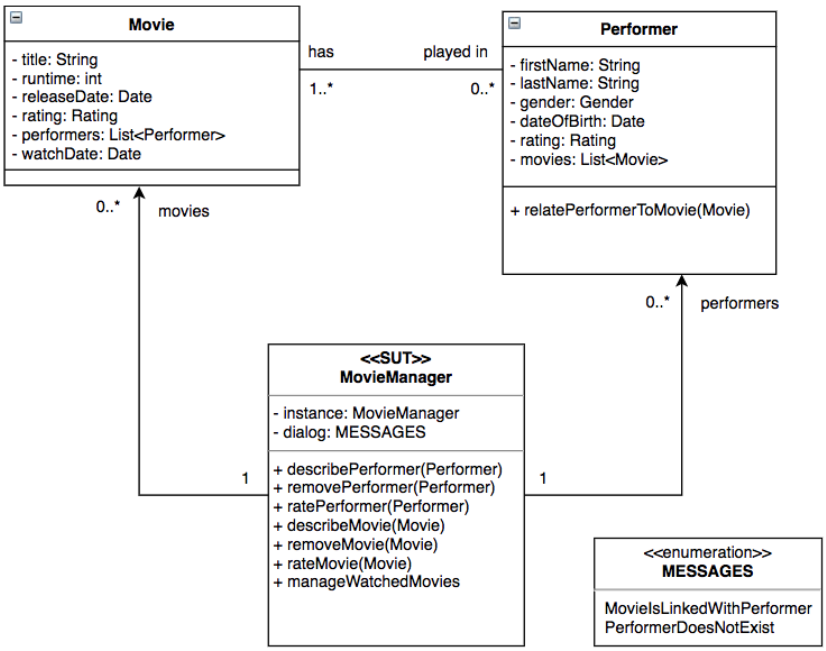

Figure 7.6: Example for a class diagram of Movie Manager

The Movie Manager Class Diagram defines the data and the points of control and observation of the application under test. The enumeration "Messages" represents the possible message/warning to be displayed on the dialog window of the application. For our case, we considered only the messages like, *movie is linked with performer* and *performer does not exist*.

In addition to class diagram, an instance diagram and a state machine are also provided in the use case scenario of the provided approach. In the approach, the test generation method of LEIRIOS Test Designer consists in testing all the possible behaviors of the specification operations, by traversing the states of the system. But in our implementation, we have to skip this step since it is performed by the LEIRIOS Test Designer tool, which is called now Smartesting as already explained in subsection 7.4.1. We cannot use the tool right now, because it requires a scheduling even for the demo version.

Therefore, we give only exemplarily two different internal activities specified according to the description in the article [12], one for the user option "remove movie", and other one for the user option "rate performer", as shown in Table 7.7.

Table 7.7: Example of internal transitions

Trigger/Label	**Guard**	**Effect**
removeMovie (movie) - Case movie is linked with a performer	Movie->linked (m: movie | m.linked=True)	MESSAGES**::** MovieIsLinkedWithPerformer **/*@REQ: MM-2@*/**
ratePerformer (performer) - Case performer does not exist	Performer->exists (p: performer | p.exists=False)	MESSAGES**::** PerformerDoesNotExist **/*@REQ: MM-7@*/**

These are the transitions, which are triggered by a user event, a guard and an effect. Requirements traceability is managed by tagging the effect of each transition with requirement identifiers. For example, the requirement MM-7 is used to annotate the effect in case of not existing performer. This is the link between the model and the requirement.
As last step, this link between the model and the requirements should be used by the LEIRIOS Test Designer tool to produce the traceability matrix between generated test cases and requirements. For each transition a test target and a test case should be via LEIRIOS Test Designer tool generated. Since we cannot use the tool, we generate the test cases as an example for both transitions manually. Table 7.8 and Table 7.9 presents these generated test cases for the two internal transitions showed in Table 7.7.

Table 7.8: Test 1: Movie is linked with performer - Covered Requirements: MM-2

Step	**Operation**	**Attributes Values**
1	MM2.removeMovie (movie)	MM2.display = MovieIsLinkedWithPerformer

Table 7.9: Test 2: Performer does not exist - Covered Requirements: MM-7

Step	**Operation**	**Attributes Values**
1	MM7.removePerformer (performer)	MM7.display = PerformerDoesNotExist

7.5 Comparison

In this section, we describe the special features, similarities and differences of each approaches.

Both approaches are very similar in terms of the way they use the information and of their steps. They both use informal requirements as input for their approaches, use system models for this purpose (UML models including OCL), and make use of requirements traceability for ensuring the quality of the generated test cases. They both offer tool support to generate tests automatically.
However, there are also differences between them. In the first approach, [2], the informal requirements are analyzed and structured into requirements models and then OCL is used for verifying the quality of these requirements models. In the second approach, [12], the expected functional requirements of the system under test are derived from a use case scenario. The requirements traceability is managed by tagging manually the postconditions of the operations and the effects of the transitions in the UML model with the requirements via ad-hoc comment symbols in order to associate a requirement with an OCL statement which will be involved into a test target during the generation of the test cases.
Furthermore, the LEIRIOS Test Designer tool provided in the second approach does not offer a tool support for tracing-back requirements from tests to models, while the first approach supports it via the automatically generated test report by their provided tool, Qtronic.

Table 7.10 presents the detailed comparison of both approaches as a synthesis matrix. In the first column, the number of the synthesis questions are presented. The synthesis questions are already documented in section 1.3. Then, the second column presents the answers to these questions for approach 1, and the third column for approach 2.

Table 7.10: Synthesis matrix

No.	Approach 1: Tracing Requirements in a Model-Based Testing Approach	Approach 2: Requirements Traceability in the Model- Based Testing Process
1a)	- Informal requirements - Requirements models via requirements diagram of SysML - UML Models, e.g. class Diagram specifies a domain model, a behavioral model describes the behavior of the SUT using state machines, data models describe the message types exchanged between different domain entities, domain configuration models represents specific test configurations using object diagrams. - Generated tests and test report	- Informal requirements - UML Models (including OCL) e.g. Use case diagram, class diagram, enumeration diagram, instance diagram, state machine. - Links between the requirements and models - Test cases and test scripts - Traceability Matrix
1b)	Informal requirements are input for this approach. A set of modeling guidelines and validation rules should also be defined. In order to automatically transform UML models to the QML via the Qtronic tool, all requirements should have been linked to model elements and the models should have been validated. The desired coverage criteria used for test generation should be manually selected from the GUI of Qtronic. The presented approach is for functional requirements and it only supports online testing.	Informal requirements are input for this approach. Uncontrollable or unobservable elements should not be modeled.
1c)	1. The informal requirements are analyzed and structured into a requirements model using the requirements diagrams of the SysML. 2. SUT is specified using UML (including OCL) and several perspectives of the SUT are modeled, e.g. class diagram, behavioral model, data models, domain configuration models etc. 3. A set of modeling guidelines and validation rules are defined using OCL. MagicDraw tool has been used for editing the SysML and UML models and for running the validation rules 4. The models used to specify the SUT are subsequently transformed into input for the automated test derivation tool Qtronic. 5. At the end of each test run, an automatically generated test report will summarize the result of the testing process in terms of generated test cases, verdicts, coverage levels etc.	1. UML-Modelling of functional requirements of the application under test, e.g. class diagrams, instance diagrams, state machine diagrams etc. 2. Tagging manually the post-conditions of the operations and the effects of the transitions in the UML model with the requirements. The format uses ad-hoc comment symbols in order to associate a requirement with an OCL statement which will be involved into a test target during the generation of the test cases. 3. Driving the test generation process in which the generation of tests is steered by the validation engineer on the basis of test selection criteria. 4. Test generation method provided by the LEIRIOS Test Designer tool 5. Executable test script generation which are converted from the generated test cases via LEIRIOS Test Designer. A test script pattern and a mapping table are to be defined by the test engineer.
2a)	Test generation, and tracing requirements to models, to test specification, and back to models again.	Test generation, and tracing requirements to test cases.
2b)	Product engineer, software engineer, requirement engineer and user of the provided tooll/tester.	Validation engineer, software engineer, requirement engineer and user of the provided tool/tester.

2c)	Chapter 1: Software Requirements - Requirements Process - Requirements Analysis - Practical Considerations (e.g. Requirements Tracing) Chapter 4: Software Testing - Test Techniques (e.g. Model-Bases Testing Techniques) - Software Testing Tools (e.g. Testing Tool Support and Categories of Tools) Chapter 9: Software Engineering Models and Methods - Modeling - Types of Models - Analysis of Models (e.g. Traceability)	Chapter 1: Software Requirements - Requirements Process (e.g. Process Actors) - Requirements Analysis - Practical Considerations (e.g. Requirements Tracing) Chapter 4: Software Testing - Test Techniques (e.g. Model-Bases Testing Techniques) - Software Testing Tools (e.g. Testing Tool Support and Categories of Tools) Chapter 9: Software Engineering Models and Methods - Modeling - Types of Models - Analysis of Models (e.g. Traceability)
3a)	The NoMagic's MagicDraw has been used for editing the SysML and UML models and for running the validation rules. Conformiq's Qtronic tool is provided for the automated test derivation and test report generation. Only online testing mode of Qtronic is used to generate tests.	LEIRIOS Test Designer tool is provided to generate test cases and to produce the traceability matrix between generated test cases and requirements. The tool is used in the test and test script generation processes.
3b)	The creation of the models is not automated and the desired coverage criteria used for test generation should be manually selected from the GUI of Qtronic. Tests and test reports are automatically generated by the provided tool, Qtronic.	Requirements traceability is managed by tagging manually the postconditions of the operations and the effects of the transitions in the UML model with the requirements via ad-hoc comment symbols. Tests are automatically generated by the provided tool, LEIRIOS.
4a)	Approach does not present an evaluation. Only validation rules have been defined and implemented for both requirements models and for SMs for checking different quality metrics of the resulting models before proceeding to the test derivation phase. For the description of the approach, only small examples from a telecommunications case study are provided.	Approach does not present an evaluation. Only for a simpler description, the approach is implemented on a simplified version of a drink vending machine controller.
4b)	Since the approach does not offer any evaluation, only the benefits and shortcomings of the approach have been shared as below: - The approach provides a solution for tracing only functional requirements and supports only online testing. - The approach provided automation of the transitions between the phases of the process, allowing to have a fast feed-back loop for testing and debugging specifications or the implementation of the SUT. - Many errors have been detected in the early stages of the process, when the system models have been created. The errors were caused mainly by omissions in the models and by misinterpreting the requirements. - Since the approach is fully automated, the effort in updating the models and performing the testing again was diminished.	Since the approach does not offer any evaluation, only the benefits and shortcomings of the approach have been shared as below: - Automatically produced traceability matrix leads to several advantages in the software testing process, e.g. it gives a clear functional coverage metrics to the generated test cases, enables to improve the model or test generation strategies, and gives valuable feed-back on the requirements. - There is no tool support for tracing-back requirements from tests to models.

7.6 Conclusion

This chapter presents two approaches for the testing with system models. Both approaches, [2] and [12], use requirements traceability in their model-based test generation processes and provide a tool support to generate tests automatically.

The literature research showed us that the for this topic relevant articles are mostly published between 2000-2009. Although there are a lot of articles which provide approaches based on the links (traces) between requirements and test cases, only a few of them are using system models. Therefore, it is important to differentiate the system model and test model with respect to the model-based testing. However, based on our research, we could not find a sharp difference between the two. Our conclusion is that test models are developed solely for testing while system models can be primarily developed for system development and then used for testing as well. Moreover, with this literature study, we came to the conclusion that it is laborious to find relevant articles with keyword-based search method, because requirements traceability and model-based testing are very comprehensive topics in Software Engineering.

The first approach studied in this chapter, [2], describes the MBT process in more detail than the second approach [12] does. But [12] presents a more detailed use case example with more visuals which enables us to understand the steps of the testing process better. However, both approaches lack the detailed explanation on how the test cases are generated via the provided test generation tools. For the generation of test cases, [2] provided Conformiq's Qtronic tool, which is unfortunately not existing anymore, and [12] provided the LEIRIOS Test Designer tool, which is named currently Smartesting. Therefore, we could not find the necessary information on how these tools implement the test generation step in practice. Unfortunately, no evaluation was offered for either approach. All of this makes it difficult for us to make an accurate assessment between these two tools/approaches. In our opinion, if we had to choose one of this approaches, it would make more sense to choose the second approach [12], as LEIRIOS Test Designer tool (Smartesting) is still on the market. It has probably evolved over the ten years as well.

Finally, both approaches presented in this chapter showed us that testing with system models enables to increase the possibility of finding errors, reduce the testing time, and improve the test quality using requirements traceability.

8 Testing Functional and Non-Functional Requirements in User Requirements Notation

8.1 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, models 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.*[4] and the article *"Transforming Workflow Models into Automated End-to-End Acceptance Test Cases" by Boucher and Mussbacher*[13]. 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 section 8.2. In section 8.3, the first paper's approach is described and applied, which will also be the case in the section 8.4 regarding the second paper. In the penultimate section 8.5, 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.

8.2 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 found 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, they are not useful to answer the research question.

Research Sources: IEEE Xplore [34], 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 8.1)

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 relevance criteria are met. Unfortunately, none 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 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*,

[1] https://ieeexplore.ieee.org/document/5475050

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, one 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 8.1: 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

**: [34] †: [3] ∇: [13]*

8.3 Approach 1: Scenario-Based Validation Beyond the User Requirements Notation

8.3.1 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 [4].

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 8.1). 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 "testable 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 case 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 by linking corresponding elements and methods. 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”.

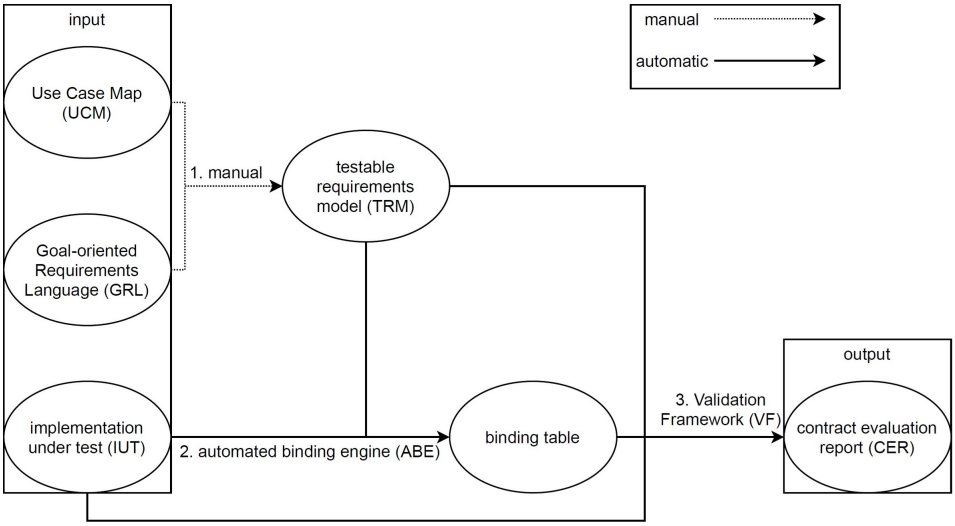

Figure 8.1: Flow of approach 1

8.3.2 Application

In this section, the approach by *Arnold et al.* applies to the Software *Movie Manager*.

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 8.2 and Figure 8.3. Figure 8.2 describes which functions can execute in the IUT. Figure 8.3 describes which goals the functions achieve and on which soft goal it has an influence.

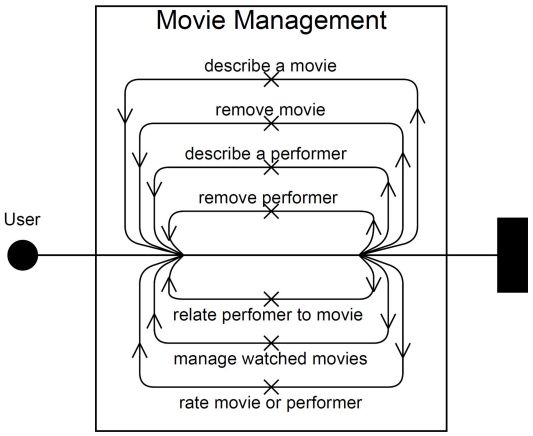

Figure 8.2: Functional Requirements in Use Case Maps Notation

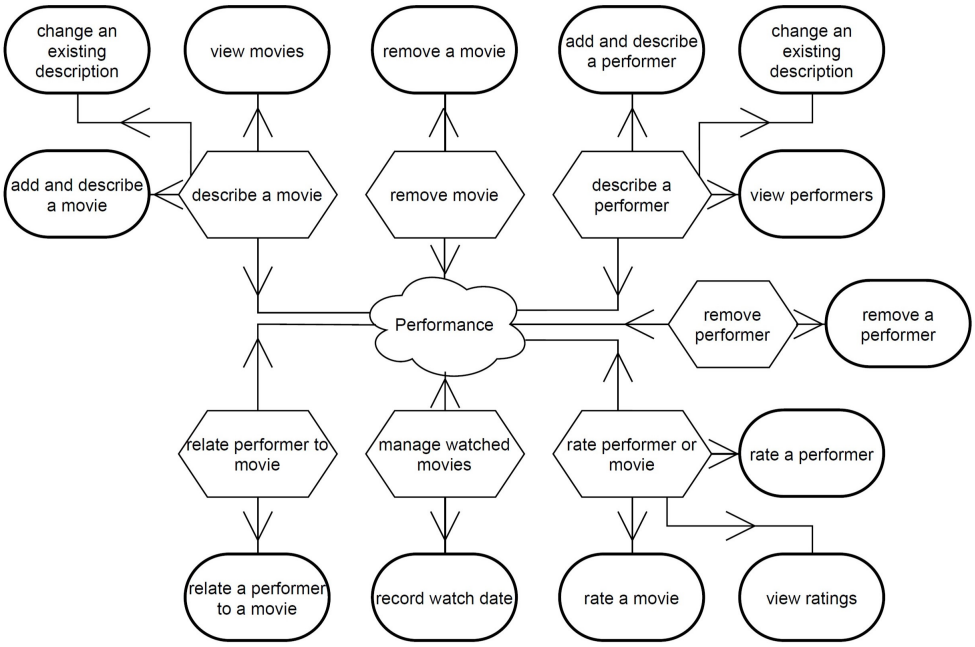

Figure 8.3: Non-functional Requirements in Goal-oriented Requirements Language

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.

For clarity, this step is split up into two. First, the functional requirements convert to the TRM. The result is in Listing 8.1. In the TRM is for each function that should be tested one responsibility. The responsibility checks the preconditions, executes the IUT’s method, updates global variables if necessary, and checks the postconditions. Help functions and help variables for the responsibilities are defined as observability and scalar.

Listing 8.1: TRM

```
Contract MovieManagment {
    Scalar Integer numberOfMovies;
    Observability Boolean HasMovie(Movie item);
    Responsibility AddMovie(Movie item) {
        Pre(HasMovie(item) == false);
        Execute();
        numberOfMovies = numberOfMovies + 1;
        Post(HasMovie(item) == true); }
    Responsibility RemoveMovie(Movie item) {
        Pre(HasMovie(item) == true);
        Execute();
        numberOfMovies = numberOfMovies - 1;
        Post(HasMovie(item) == false); }
    Exports {
        Type Movie { not context; } }
}
```

Second, the non-functional requirements must be added to Listing 8.1. The result is in Listing 8.2. Two new scalars, *add_timer* and *remove_timer*, are needed to evaluate the performance. This timer must start before and stop after the execution in the responsibility. Also, two scenarios are needed. *AddRemoveMovie* define the behavior of an element in the *Contract* and the *Lifetime* define the lifetime of the *Contract*. *Reports* write the information in the CER and need the two *Metric's TimesToAddMovie* and *TimesToRemoveMovie*.

Listing 8.2: TRM

```
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() {
```

```
27          add_timer.Values();  }
28      Metric List Integer TimesToRemoveMovie() {
29          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  }
```

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 8.3. The contract, all export types, all responsibilities, and all observabilities from the TRM need to link to the corresponding elements from the IUT. In the first column are the TRM elements, and in the third column the IUT elements.

Table 8.3: 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.

8.4 Approach 2: Transforming Workflow Models into Automated End-to-End Acceptance Test Cases

8.4.1 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) [13].

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 (see Figure 8.4). 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 skeleton, 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 according to the paper 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 of the paper [13], 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.

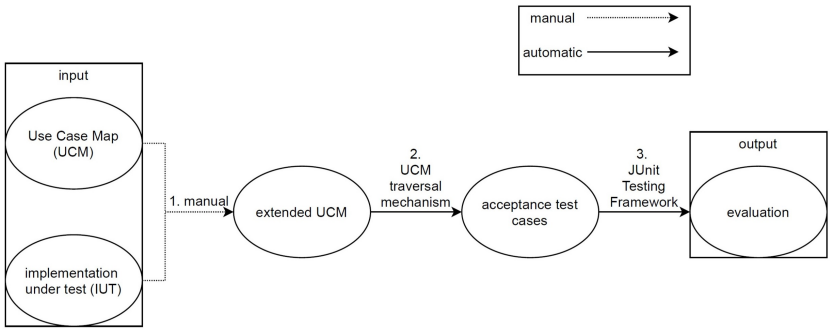

Figure 8.4: Flow of approach 2

8.4.2 Application

In this section, the approach by *Boucher and Mussbacher* applies to the Software *Movie Manager*.

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 Figure 8.5. Figure 8.5 describes which functions can execute in the IUT.

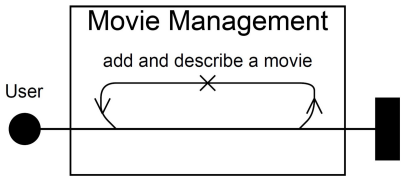

Figure 8.5: Functional Requirements in Use Case Maps Notation

First Step: Extend the UCM

The extended UCM is on view in Figure 8.6. The UCM from Figure 8.5 is defined with information about the input data, the expected output, a description of the behavior, and the test logic with postconditions. In this case, the function *createMovie* must get as input two strings for name and country, which are not none and not empty. The generation of the input combination is described in subsection 8.4.1

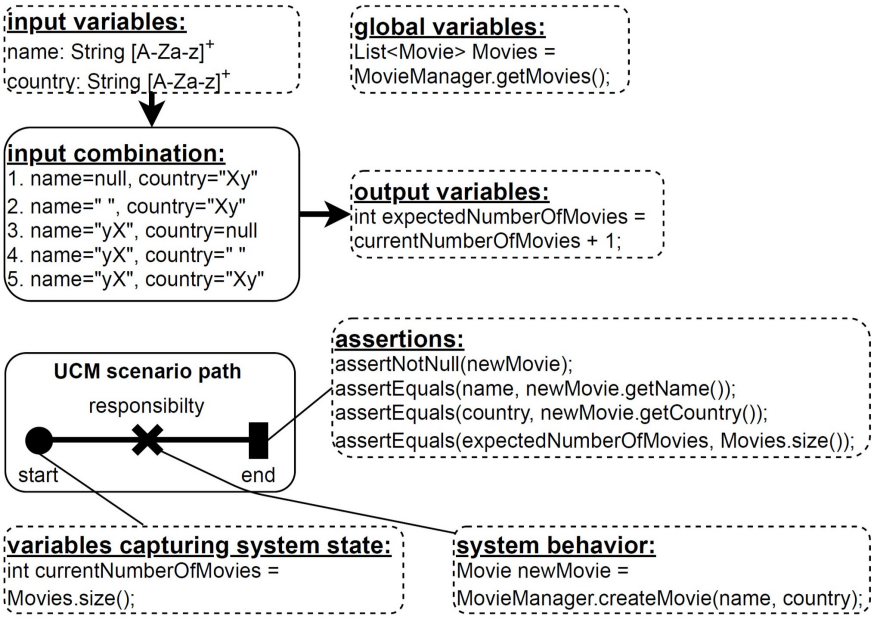

Figure 8.6: Extended Use Case Map

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 8.3. The traversal mechanism traveling through the Use Case Map from Figure 8.6, from the start point to an endpoint by visiting the responsibilities and create automatically a test case skeleton which is used with the input combinations to create the specific acceptance test cases. In this case, four invalids and one valid test case.

Listing 8.3: JUnit test cases

```
public class MovieManager {
    //global variables
    List<Movie> Movies = MovieManager.getMovies();
    @BeforeClass //method TODO
    @Before //method TODO
    @After //method TODO
    @AfterClass //method TODO
    @Test
    public void nameNullTest() {
        String name = null;
        String country = "Xy";
        int currentNumberOfMovies = Movies.size();
        try{
            Movie newMovie = MovieManager.createMovie(name, country);
            fail();
        } catch (Exception exception) {} }
    @Test
    public void nameEmptyTest() {
        String name = "";
        String country = "Xy";
        int currentNumberOfMovies = Movies.size();
        try{
            Movie newMovie = MovieManager.createMovie(name, country);
            fail();
        } catch (Exception exception) {} }
    @Test
    public void countryNullTest() {
        String name = "yX";
        String country = null;
        int currentNumberOfMovies = Movies.size();
        try{
            Movie newMovie = MovieManager.createMovie(name, country);
            fail();
        } catch (Exception exception) {} }
    @Test
    public void countryEmptyTest() {
        String name = "yX";
        String country = "";
        int currentNumberOfMovies = Movies.size();
        try{
            Movie newMovie = MovieManager.createMovie(name, country);
            fail();
        } catch (Exception exception) {} }
    @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  }
```

Third Step: Run JUnit Testing Framework

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

8.5 Comparison

The approaches have been compared using a set of synthesis questions, as shown in Table 8.4 and Table 8.6

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 Testable Requirements Model. In the second, the Use Case Maps must be extended. In theory, the transfer to the Testable 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 Testable 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 8.4: Synthesis Matrix part 1/2.

No.	Approach 1: Scenario-Based Validation Beyond the User Requirements Notation	Approach 2: Transforming Workflow Models into Automated End-to-End Acceptance Test Cases
1a)	- "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". - "Testable 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. - "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.
1b)	- FRs and NFRs in URN - IUT	- FRs in URN
1c)	- The requirements must be converted 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.
2a)	- 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.
2b)	- 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.
2c)	- 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 Software Engineering Models and Methods: 1.4 Preconditions, Postconditions and Invariants

Table 8.6: Synthesis Matrix part 2/2.

No.	Approach 1: Scenario-Based Validation Beyond the User Requirements Notation	Approach 2: Transforming Workflow Models into Automated End-to-End Acceptance Test Cases
3a)	- "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.	- "UCM traversal mechanism" from "jUCMNav tool"$^{\nabla}$: Traverse the extended UCM and create the acceptance test cases. - "JUnit Testing Framework": Execute the acceptance tests on the software.†
3b)	- 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 automatically.
4a)	- The authors did not evaluate the approach.	- They have compared the lines of codes of the extending process with the lines of codes of the manually generated test cases. This was made on example software.
4b)	- The authors did not evaluate the approach.	- They need 60 lines of code to extend the UCM and 600 lines of code 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/

8.6 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[2] from the paper to the website is not available. However, with "Wayback Machine - Internet Archive"[3] a copy of the website[4] 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[5] and JUnit[6], 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"*.

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

9 Testing Non-Functional Requirements with Risk Analysis

9.1 Introduction

In contrast to functional requirements (FRs) that describe the program's functionality, i.e. how it processes data and user input, non-functional requirements (NFRs) describe constraints that the program must adhere to [16]. Parts of NFRs are performance and security requirements which can directly affect the end-user but also maintainability requirements which are more important to development teams. While there are lot of resources about testing FRs in the form of unit-, integration- and end-to-end-tests, no common testing framework or guidance for testing NFRs exists.

For this reason, we look into two approaches [61] and [39]. While [61] was given in advance by the advisors of the seminar, [39] was found through a literature search, which is described in section 9.2. Each approach will be described and applied to an example in the form of movie management software in the respective section 9.3 and section 9.4.

Both approaches are compared in section 9.5 by using a synthesis matrix. We will look at which NFRs are covered and how they are tested. The results of this chapter will be summarized and concluded in section 9.6.

Please refer to the glossary for the following terms used throughout this chapter: test case, use case, NFR.

9.2 Literature search

The starting point for the literature search was the paper given to us [61]. Based on this paper, we formulated the central research question:

> "*Which approaches for testing non-functional requirements systematically with risk analysis exist?*".

We focused on finding articles that covered the three most important keywords and phrases for the topic: *testing*, *non-functional requirements* and *risk analysis*. A quick search using these three phrases resulted in IEEE Xplore having the most promising results, whereas ACM [3] only showed a few. Because the given article [61] can also be found on IEEE Xplore [34], we focused our search onto that site but still looked at ACM.

To be able to evaluate the relevance of papers found during the literature search, we defined three relevance criteria:

1. Does the article cover non-functional requirements? They must not only be mentioned as a side note next to functional requirements.

2. Does the article combine risk analysis with tests?

3. Does the article cover *testing* of non-functional requirements?

Even though these relevance criteria basically only cover the research question, they filter out most non-relevant papers as we will see later on.

The search was carried out by using forward and backward snowballing as well as by using search terms with combinatorial modifiers. Only two papers reference [61] according to its IEEE Xplore site, both of which cover functional but not non-functional requirements. The paper itself references 23 papers. Of those papers, only few covered the first criterion and none covered the third criterion. [61] itself does not cover risk analysis as a main research topic but only covers it in a side note (see section 9.3). This is why search-term based search was performed using the key terms: non-functional requirements, testing and risk analysis.

Table 9.1: Term based search results

Source	**Date**	**Search query and restrictions**	**#Results (relevant)**
IEEE Xplore	2020-11-11	`"non-functional requirements" AND testing AND "risk analysis"`	3 (0)
IEEE Xplore	2020-11-11	`non-functional requirements AND testing AND risk analysis`	12 (0)
IEEE Xplore	2020-11-11	`risk AND "non-functional" and test`	23 (2)
IEEE Xplore	2020-11-29	`(("Abstract":nonfunctional requirements) AND "Abstract":Test) AND "Full Text & Metadata":"risk analysis")`	2 (1)
IEEE Xplore	2020-11-29	`(("Abstract":non-functional requirements) AND "Abstract":Test) AND "Full Text & Metadata":"risk analysis")`	3 (0)
ACM	2020-11-29	`[Abstract: "risk"] AND [Abstract: test*] AND [Abstract: "non functional"]`	4 (1)
ACM	2020-11-29	`[Abstract: test] AND [Abstract: "non functional"] AND [[Full Text: "risk analysis"] OR [Full Text: "risk"]]`	20 (1)

Table 9.1 lists an excerpt of the term-based search. Listed are only those searches that returned promising results or highlight issues I encountered during the search. It can be seen that, if all keywords are combined using the `AND` operator with the default restrictions, no relevant results were returned. After a feedback from one advisor, the search was changed so that "non-functional requirements" and "testing" were expected in the paper's abstract, but "risk

analysis" was searched for in *all* metadata including the full text. It turned out that no papers were found which mentioned risk analysis as well as the other two keywords in their abstract.

We also discovered that the spelling of the term "non-functional" had a huge impact on the results returned by IEEE Xplore.

After this initial search, the resulting papers were evaluated and one paper was chosen. The papers to chose from included:

- "Scenario-Based Assessment of non-functional Requirements " [29]
- "Alignment of requirements specification and testing: A systematic mapping study" [9]
- "Using Automated Tests for Communicating and Verifying Non-functional Requirements" [39]

Scenario-Based Assessment of non-functional Requirements covers all three criteria we defined at the start of our search. However, it limits itself to complex socio-technical systems and only looks at one non-functional requirement, which is the system's performance. It limits itself to the evaluation of the reliability of certain aspects of software which interacts with humans to calculate the risk of human errors occurring. This is done by implementing scenarios–hence the title "scenario-based assessment". The testing aspect of this paper is limited to human-interactions whose risks are evaluated. If scenarios fail this risk assessment, then so will the test.

Alignment of requirements specification and testing: A systematic mapping study is about papers that cover non-functional requirements. It is a study about such papers and lists approaches that are used to test NFRs. Some of which are mentioned in other chapters of this paper. However, none cover risk analysis. The paper itself does not give much insight into testing non-functional requirements itself.

The chosen article which we will further evaluate in the following sections is *Using Automated Tests for Communicating and Verifying Non-functional Requirements*. It covers the non-functional requirement "maintainability" and how it can be tested. It further explains it by using practical examples. However, the chosen paper does not cover risk-analysis. Since no paper could be found that covers all criteria, we were advised to focus on the testing of non-functional requirements and leave out risk-analysis.

9.3 Approach 1: Control Cases during the Software Development Life-Cycle

9.3.1 Description

In their paper "Control Cases during the Software Development Life-Cycle" [61], J. Zou und C. J. Pavlovski based their work on so called "control cases" and "operating conditions" as tools for modeling and controlling NFRs.

"Control cases" are used as a format to communicate and discuss NFRs between management, requirement engineers, developers, and system users and to define qualitative attributes of the system.

Their work focuses on determining and revealing problems early on, for example bad performance or security risks. The classic software development life cycle often focuses on these topics too late or not at all. However, control cases require NFRs to be defined first. To define them, the paper starts by introducing operating conditions. Operating conditions model constraints that apply to the system or a specific use case. These constraints are then used to model NFRs, hence the operating condition can be seen as a high level view on NFRs. Defining such constraints that apply to a certain use case is left to the reader or rather is mentioned as a step of the business process modeling.

Operating conditions can belong to one or more use cases and are not unique to a specific one. Conditions such as "Transaction Volume Condition: >400 concurrent users" can be applied to different use cases [61] and model exactly that: a condition under which the use case operates.

Control cases–as the name suggests–control the operating conditions and can be used to ensure that they are complied to. This makes it possible to mitigate business risks which may affect the business if the operating condition and its constraints are violated. Their paper visualizes the connection between these artifacts using an UML diagram, which can be seen in Figure 9.1 below.

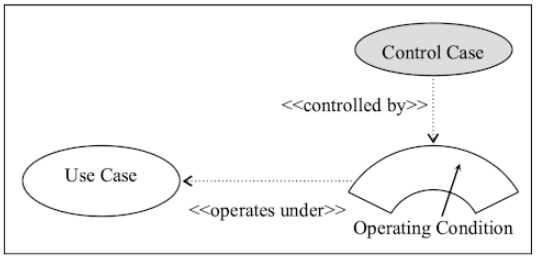

Figure 9.1: Association in Use Cases Modelling [61]

The paper creates such a control case by introducing a fictional example of a traveling agent. All of the previously mentioned artifacts are created during the "Business Process Modeling" and are refined throughout the software development life cycle. This means that operating conditions and control cases are defined together with use cases and can be incorporated together in a use case model. The paper does this for their fictional example which can be seen in Figure 9.2.

In this graphic, control cases are visualized as shaded ellipses and operating conditions as speedometers, though unspecified by the paper. This graphic also emphasizes that operating conditions are not bound to one specific use case but can be applied to different ones. And the control case is specific to one operating condition.

The reader is guided through all steps of the software development life cycle, so that a control case can be defined which is then used as the basis for a test case. Because the control case is associated to an operating condition, the test case is associated to it transitively as well. The test case exists to verify that the controls put in place to manage the operating condition are effective.

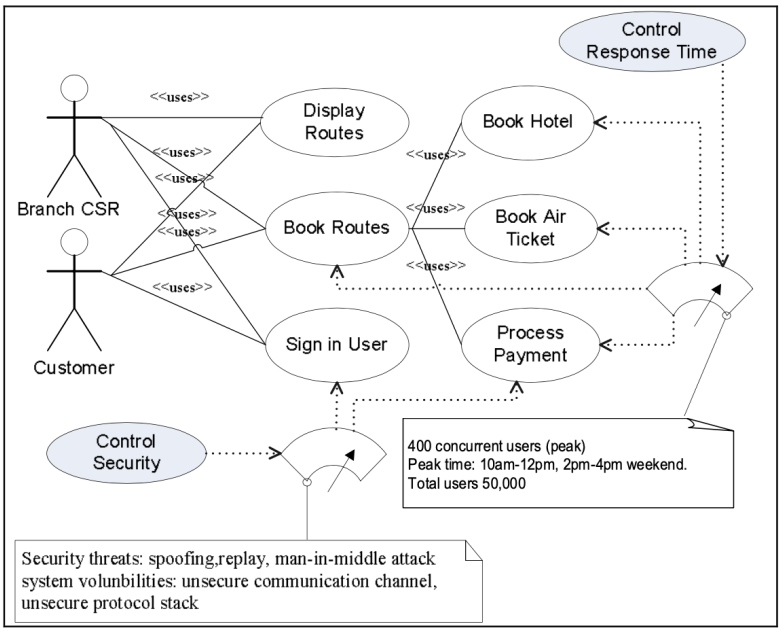

Figure 9.2: Use Case Model with Control Cases [61]

The paper does not give a detailed instruction how to model test cases. It only instructs testers to simulate the operating condition, for example by creating a huge work load on the server. With this simulation relevant metrics can be extracted that are used to verify the test case.

9.3.2 Application

J. Zou und C. J. Pavlovski focus on creating control cases. This is done for the movie manager example.

We first define one goal of our software: it must contain a movie list view that has smooth scrolling and can handle a large amount of movies. This also defines a constraint and therefore our operating condition: we must operate a smooth list view. If this cannot be accomplished an associated risk may affect the business. The control case bundles all of this in a matrix which is defined as in Figure 9.3 on page 104.

Based on this control case, developers can start to implement the software. During testing stage, functional requirements can be tested by basing them on use cases. Non-functional requirements, on the other hand, can be tested by creating tests from control cases. The tester needs to simulate the operating condition. In our example above, that would mean to simulate the scrolling condition by creating a huge list of movies. The steps that must be executed for the test are combined into a test case , such as Figure 9.4 on page 105.

By determining the operating condition that is associated to a use case, we were able to create a control case that reflects the NFRs. Based on the operating condition, we then created a test case that checks if the controls put in place by the control cases are enough to mitigate the business risk. Following this pattern, tests for non-functional requirements can be created systematically step by step.

Control Case: Performance of the movie list view
Control Case ID: CC-001
Operating Condition: Scrolling Speed Condition
Description: The control case describes the "smoothness" while scrolling through the movie list view. Scrolling must be smooth. If it is not then users may assume bad performance.
NFR Category: Performance and Capacity
Associated Use Cases: Show movies in list view
Technical Constraints: GUI Framework, Operating System (e.g. 32bit system only allows addressing of 4GB main memory)
Vulnerability: Unknown number of movies. Users may only have a few or thousands of movies. Analyzing movies (or doing other work) must not lead to the movie list view being unresponsive. Having a lot of movies must not make the program run out of memory.
Threat Source: None (local software used by one user)
Operating Condition: There may be tens of thousands of movies. Assuming that each movie object has a size of 600kB (only meta data and a small thumbnail), loading 20,000 movies would lead up to 12GB of memory usage[1]. All movies must be represented in a list view.
Business Risk: If scrolling is not smooth, the user may switch to other software or leave a bad rating.
Probability: medium (likely few users are affected)
Risk Estimation: low (users with huge databases may accept higher load times or sluggishness in the UI)
Control: 1. Only load visible movies into main memory. Use "infinite scrolling" techniques. Remove those movies from main memory that are not visible to the user. 2. Only load the title into main memory. Load other details only if required. This reduces the memory footprint.

Figure 9.3: Control Case for Approach 1 of Topic 9

Associated Control Case ID: CC-001
Test Objectives: Verify that the movie list view has no visible hiccups when scrolling through the list of movies.
Preconditions: Movie manager is up and running.
Test Steps: 1. Create 20.000 movies and load them into the movie manager 2. Open the movie list view 3. Scroll through the list of movies
Expected Result: 1. The end of the list view is reached. 2. No hiccups while scrolling were visible, i.e. no "sluggishness".
Notes: The test must be performed on a system that has at most 8 GB of RAM to reflect common end-user hardware.
Test Result: Pass / Fail

Figure 9.4: Test Case for the movie manager example of topic 9, approach 1

9.4 Approach 2: Using Automated Tests for Communicating and Verifying Non-functional Requirements

9.4.1 Description

In "Using Automated Tests for Communicating and Verifying Non-functional Requirements" [39], Robert Lagerstedt describes how testing NFRs can be automated by introducing a tool-based approach. The author only looks at NFRs in regards to software architecture which affects code quality in the sense of maintainability and security.

By looking at software architecture aspects as NFRs, Lagerstedt describes how software may be written by listing some architectural requirements. It should not have dependencies from lower code components into higher but only vice versa. Certain functions must not be called from some components to ensure encapsulation. Some functions may be blacklisted due to security concerns. All of these requirements are part of the software architecture and therefore a huge part of software quality and maintainability [39].

These NFRs must be communicated to developers. According to Lagerstedt, this is done by guidelines written by software architects. The compliance of these guidelines is often verified by different reports. These reports may be written for each code change as part of a code review or by other teams. Lagerstedt visualizes this in a simple UML diagram as can be seen in Figure 9.5. The graphic uses a rather high distance between the developer and the compliance

report on purpose to symbolize that the two are asynchronous, this means that the report is not automated and feedback reaches the developer not immediately.

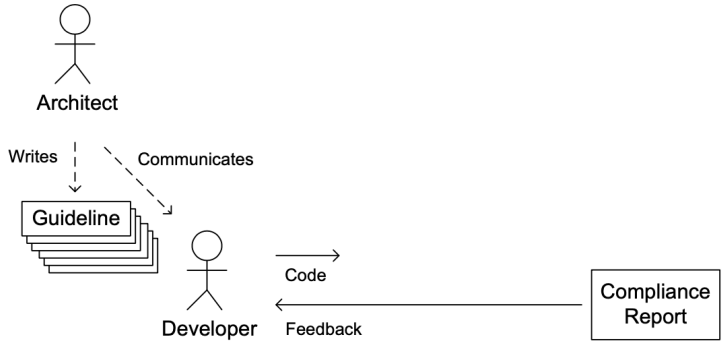

Figure 9.5: The common way of communicating architectural requirements [39]

This way of communicating guidelines is not very cost-efficient. Every developer has to read and understand the guidelines. Developers must be re-trained when changes are made or if too many guidelines violations occur, because they have been forgotten. This is quite time consuming and prone to error. Creating reports about guideline compliance is time consuming as well. Furthermore, while code review should be performed for all code changes, mistakes may slip through.

That is why the author proposes automated testing of software architecture NFRs. This allows a fast tool-based feedback loop in which the developer gets a code review that can be incorporated without other developers having to look out for violations of guidelines. On top of that, by having this tight feedback loop, developers can learn the guidelines in an iterative way. Little to no training is required, which saves time to make new guidelines known to all developers.

The guidelines are written as tests. These tests can be included in existing static code analysis tools such as linters and other code checkers. Developers can see the results of such tools. Furthermore guidelines are communicated to the developer in case of a test failure. Lagerstedt uses Figure 9.6 to visualize this approach. Developers get feedback through different tools that the architect extends. Tools such as the editor, compiler or static analysis tools.

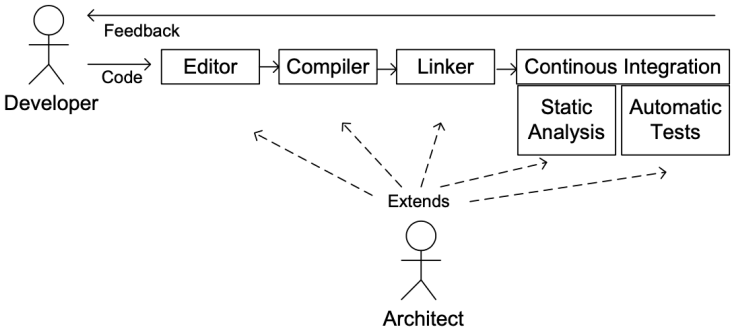

Figure 9.6: Suggested solution of communicating requirements [39]

According to Lagerstedt's personal experience, a tool based approach is superior to a guideline-only one. Productivity is increased while the time spent on communicating architectural guidelines is decreased. The number of non-compliant code is lower for the tool-based approach than for using guidelines and reports only.

9.4.2 Application

The paper works with architectural NFRs but does not explain how those can be modeled. To be able to apply the approach, we introduce another system function to the movie manager example that is listed in Table 9.2. This system function and the following NFRs are based on personal experience in maintaining an open-source media manager.

Table 9.2: New system function for application of approach 2 of topic 9

Name	Export the movie to HTML
Description	An existing movie is exported to a single HTML file which can be viewed in any modern web browser
Precondition	Movie exists
Input	Movie details
Postcondition	HTML file exists with the movie's contents
Output	HTML file

In Table 9.3 on page 108, two NFRs are listed which were created for the system function in Table 9.2. These two NFRs are based on personal experience. Both are transformed into pseudo code so that the NFRs can be executed automatically as part of the code review.

While these two NFRs can be written as guidelines, especially point two may be violated and may slip through code review. Violating point two may result in security issues or at least in unexpected behavior if the HTML contains unescaped characters.

[1] From personal experience by maintaining a media manager. Users regularly report more than 10,000 movies in their database.

Table 9.3: NFRs for the application of approach 2 of topic 9

No.	NFR	Explanation

1	IMDb IDs are encapsulated in a class	All IMDb IDs have a certain format. They start with the string "tt" and end with 7-8 numbers. The ID must be validated which cannot be ensured by using a simple string. This is why an encapsulation in a class is required.
		Furthermore the programming language's type system can help to identify conversion bugs as well.
		
		Implementation in pseudo code
		
		`for each $variable in $source:`
		`  if $variable.startsWith("imdb") then`
		`   if typeof($variable) != "ImdbId" then`
		`    throw new Exception("Wrong class")`

2	Exported strings are escaped	This is a security concern and can be implemented in different ways. We assume that an HTML-exporter was created which takes a movie object as an argument. This object may contain texts which contain HTML elements themselves. These elements need to be escaped. To ensure this NFR, all strings must be run through a certain function which escapes strings. Because this may be missed by the developer, a new string-subclass is introduced which escapes its input automatically, e.g. `EscapedString`. Only this string class may be used in the HTML exporter.
		
		Implementation in pseudo code
		
		`for each $functionCall in $HTMLExporter:`
		`  if $functionCall == "writeText" then`
		`   $arg = argument of($functionCall);`
		`   if typeof($arg) != "EscapedString" then`
		`    throw new Exception("Wrong class")`
		
		Note: We assume that `writeText` is a method of a generic HTML-class which the HTML-exporter uses itself. We assume that the method cannot be changed to accept another argument type. Otherwise the language's type checker could already be able to find this issue.

9.5 Comparison

For an improved comparison of these two approaches, a synthesis matrix is provided which references the questions from section 1.3:

Table 9.4: Synthesis matrix

No.	Approach 1 [61]	Approach 2 [39]
1a)	Operating conditions are formed that work under specific use cases. These, on the other, hand are controlled by control cases and can be operated under them. Control cases describe the business risks in case that the operating condition cannot be fulfilled. Because use cases and control cases are tightly connected to each other, they can be modeled in one consolidated model.	Coding guidelines are written and transformed into tests that can be used by tools in code review. These point out issues that the developer can fix. Guidelines are characterized by the fact that they describe the code architecture.
1b)	There are no preconditions because we start defining control cases at the beginning of the software development process, for example at the "Business Process Modelling"-step.	The guidelines cover code architecture. They must be transformable into automated tests (e.g. by a static code analyzer).
1c)	Preconditions/Constraints must be extracted from which NFRs are created, e.g. performance or security constraints. These constraints are modeled by operating conditions for which control cases are created. Their purpose is to mitigate business risk which is essentially the failure to fulfill the operating condition. For each control case a test case is added that checks if the controls put in place by the control case are effective. The test case basically recreates the operating condition, for example by using stress testing.	Code guidelines such as naming conventions or prohibited function-calls are defined. These are transformed into automated tests that can be executed by the developer (i.e. a tool based approach). The exact process is not explained and it is left to the reader how this may be implemented. It is only pointed out that existing tools such as compilers or static code analysis tools can be extended and used.
2a)	Early modeling of non-functional requirements. Being able to control requirements throughout the whole software development life cycle.	Maintainability, quality and security of the code base can be hold up to standards and may even be improved by giving automated feedback that points out NFRs which are violated by the developer.
2b)	Management, Requirements Engineer, Developers, Testers	Developer / test writer, Software Architect
2c)	Software Requirements (functional and non-functional requirements, acceptance tests), Software Testing (model based techniques)	Software Testing (Software Testing Tools, Test Techniques), Software Maintenance (Software Maintenance Tools)
3a)	No tool support for generating "Control Case"-Boxes and other artifacts	Existing static code analysis tools (e.g. linters), which can be extended by further tests.

3b)	No automation is done. Automation is only proposed as another step which can be implemented, e.g. through code generation with SysML.	Only code testing is performed automatically. And only tests for NFRs which were extracted from the software architects guidelines and that were transformed into automated tests. Those tests can be executed automatically during code review, e.g. by a continuous-integration service which tests each code change. Writing the tests is still a manual job.
4a)	The approach was explained by creating a fictional example and going through all steps of the software development life cycle by extending the example. No evaluation was performed, though.	No evaluation was performed. The conclusion, i.e. success of the approach, is based on personal experience only.
4b)	N/A	Based on his experience in both small and large organizations, Lagerstedt concludes that automated verification of non-functional requirements by using tool-chain feedback is superior to classic guidelines that need to be checked by humans. By evaluation of his prior experience, he concludes increased productivity and a decrease in time spent on communicating architectural requirements.

If we compare the two papers using the synthesis matrix above, we notice that they do not share a lot. That is not surprising: the second paper is very specific and only deals with architectural NFRs in code. The first paper, on the other hand, can be applied to different NFRs, not limiting itself to a specific one. Only the first paper mentions risk analysis but only as part of a control case.

Both papers do not give specific instructions how test cases can be modeled. While the first paper only says to "simulate the operating condition" [61], it leaves out details. For example security NFRs are explicitly mentioned but it is left out how an operating condition for that NFR can be simulated. Also the example test case from the paper is essentially a stress test. The second paper leaves it to the reader to develop automated tests and only mentions that static code analysis tools can be used.

While the second paper talks about test automation, it does not talk about creating tests automatically but rather about running them automatically 39. The first paper does not include any automation step at all. Neither for creating test cases automatically nor for running them.

Both do not include an evaluation of their results besides personal experience. The first article states no evaluation at all and only discusses the approach for defining the control case.

9.6 Conclusion

Both articles deal with NFRs. While [61] describes how these can be defined and controlled, it does not specify a way to test them except for simulating the operating condition. In the same way there is no description of how the business risk affects the test case except for defining the test-priority. However, it explains in great detail how control cases and operating conditions can be defined and how they interact with use cases and functional requirements, which raised my interest in the overall topic of testing NFRs. But the lack of detailed explanation for test case creation makes it difficult for me to assess the usefulness of the approach. After reading the paper I may know how to model NFRs with operating conditions but still wonder how they can be properly tested.

[39] on the other hand leaves it to software architects to define NFRs. The paper only uses architectural NFRs that exist as code conventions and other guidelines. The author describes why having automated tests is a necessity of software development in regards to cost efficiency and how it mitigates human error during code review which can be seen as a risk to code maintainability. This corresponds to my personal experience. By using a code formatter, the amount of formatting related review comments went down to zero. By introducing a new linter rule, I was able to automatically fix company branding issues in product messages which none of my colleagues were even aware of. I can therefore only emphasize that communication of NFRs is more effective and efficient when a tool based approach is used.

Finally, both articles mention risk analysis only as a side note, if mentioned at all. It is left to the reader where risks are mitigated. The conclusion is that NFRs with higher risks need to be paid more attention to by giving the tests higher priority.

10 Testing Non-Functional requirements with Aspects

10.1 Introduction

Software testing ensures that specified requirements, functional and non-functional, are met by the implementation. Non-functional requirements are especially hard to test because they do not describe what the software does, but how and to what extend of quality it does it. Hence the also customary term quality requirements. They pose numerous challenges for software testing and software quality due to their system-wide effects and often crosscutting nature. They do not only concern one part of the software and its source code, but multiple or even the entire system. For instance, the memory requirement: "The system only uses two gigabytes of main memory at most at all times." has a restrictive influence on other requirements and system functionality like performance requirements. This impedes the observance of the widely propagated principle of separation of concerns, introduced by Parnas [49] and Dijkstra [21], throughout development and testing.

By virtue of the afore-mentioned problems, there are only few tools and systematic approaches for testing non-functional requirements. Frequently, they are evaluated subjectively, resulting in a loss of traceability between test code, source code and requirements. Aspect orientation is a technique that can be harnessed for testing non-functional requirements. It is a programming paradigm, aiming to modularise such requirements (in the context of AOP, they are called system-level concerns in contrast to core-level concerns), similar to the modularisation of functional requirements using objects. An aspect modularises a system-level concern and thereby represents a system-wide functionality, accessible to multiple classes and other parts of the software. Let us consider a common application for AOP: Listing 10.1 shows a simple logging aspect in C++. It consists of three main components:

- Joinpoint: The point in the program code, the aspect is executed. It can be before, after or around something (here: before).
- Advice: Associates the joinpoint with an activity (here: a printf statement).
- Pointcut: Selects a suitable joinpoint out of the set of all joinpoints and associates it with a method or function (here: %::%()).

Listing 10.1: Logging Aspect in C++.

```
aspect LoggingAspect{
   public:
      pointcut logMethods() = call("% %::%()");
      advice logMethods() : before(){
         printf("> Enter: %s\n", JoinPoint::signature());
      }
};
```

In this example, the printf logging statement is executed each time before the %::%() method is called. Based on these principles, many possibilities for systematic and automated testing are conceivable.

In the following, the execution and results of a literature search based on a given article are presented in section 10.2. Subsequently, the given article , which assesses the use of aspects for testing non-functional requirements as well as the suitability of certain non-functional requirements for aspect-oriented testing is described and applied to an example in section 10.3. Likewise, for a selected article found using the literature search in section 10.4. It expands upon the basic ideas of the first article and partially automates the creation of test aspects. Thereafter, the results of a literature comparison are presented in section 10.5. Finally, in section 10.6 both approaches as well as the general concept of testing using aspects are evaluated.

Please visit the glossary to get familiar with the following terminology: aspect, AspectC++, AspectJ, aspect oriented programming (AOP), separation of concerns, domain specific language (DSL), non-functional requirements (NFRs) and test objective.

10.2 Literature search

To find relevant literature, a literature research based on the following search question was conducted: ***Which approaches for systematic creation of tests (for non-functional requirements) using aspects exists?*** Both forward and backward snowballing proceeding from the given article as well as a term-based search were carried out. The used terms were test, aspects, AOP and aspect-oriented. Restrictions to reduce the number of hits are described in Table 10.1. First, the upper term from Table 10.1 with everything in the publication title was used, then the lower term from Table 10.1 with test in the title, aspects in the abstract and AOP and aspect-oriented in all meta data. Source platforms for the search were IEEE [34] and ACM [3]. IEEE because the given article as well as articles referencing it can be found here. ACM because there are many available and peer-reviewed articles differing from IEEE. Content-based relevance criteria were derived directly from the search question:

- The article must cover the creation of tests using aspects because this is the main topic of the given article. Furthermore, the criterion is used to exclude articles covering testing of aspect-oriented software with conventional methods.
- The article must describe systematic approaches for the creation of tests, since this is the superordinate topic of the seminar.
- The article must describe test methods for non-functional requirements, as this is the subtopic of the given article.

In addition to these content-based criteria, there were some rather soft criteria:

- The article must be from different authors.
- The article must be written in English or German language.

Table 10.1: Search Terms, Restrictions and Sources.

Term	**Restrictions**	**Sources**
test AND (**aspects** OR **AOP** OR **aspect-oriented**)	• all in title	• IEEE • ACM
test AND **aspects** AND **AOP** AND **aspect-oriented**	• test in title • aspects in abstract	• IEEE • ACM

Table 10.2 shows the execution and results of the literature search. For further selection, the relevant articles were divided into three categories:

- overviews of testing using aspects,
- examples for testing using aspects,
- improvement or monitoring of conventional (unit) test using aspects.

The given article can be classified between overview and example. Via the literature search, I wanted to find an article providing automation and tool support, seen as the given article has shortcomings in that regard. Moreover, all relevance criteria must be fulfilled and it would be beneficial if the article covers at least two out of three classification categories.

Many articles violate the first criteria, because they cover the testing of aspect-oriented software without using aspects. Some articles were not chosen because they just cover one classification category, therefore not really adding new information and approaches.

The article that stood out was Duclos et al.: "ACRE: An Automated Aspect Creator for Testing C++ Applications" [22] because it covers all categories and satisfies all criteria. It includes an extensive general section followed by the description of an approach for automated creation of test aspects on an example to improve or replace conventional tests. Since the article was found using forward snowballing, it directly references the given article and proposes solutions for problems the latter raises. This article was selected.

Table 10.2: Literature Research Documentation.

Source	**Date**	**Restrictions**	**Term**	**Results**	**Relevant**	**Used**	**Comments**
IEEE*	11.11.2020	none	forward snowballing	10	5	∇	-
IEEE*	11.11.2020	none	backward snowballing	22	5	none	-
IEEE*	11.11.2020	none	"All Metadata":test AND ("All Metadata": aspects OR "All Metadata":AOP OR "All Metadata":aspect-oriented)	220,605	?	none	too general, not considered
ACM†	11.11.2020	none	[All: test] AND [[All: aspects] OR [All: aop] OR [All: aspect-oriented]]	172,120	?	none	too general, not considered
IEEE*	11.11.2020	title	"Document Title":test AND ("Document Title": aspects OR "Document Title":AOP OR "Document Title":aspect-oriented)	158	11	none	first 50 considered
ACM†	12.11.2020	title	[Publication Title: test] AND [[Publication Title: aspects] OR [Publication Title: aop] OR [Publication Title: aspect-oriented]]	311	4	none	first 50 considered
IEEE*	12.11.2020	test: title, aspects: abstract	((("Document Title":test) AND "Abstract":aspects) AND "All Metadata":AOP) AND "All Metadata":aspect-oriented	33	16	none	-
ACM†	12.11.2020	test: title, aspects: abstract	[Publication Title: test] AND [Abstract: aspects] AND [All: aop] AND [All: aspect-oriented]	31	6	none	-

**: [34]* †*: [3]* ∇*: [22]*

10.3 Approach 1: "Testing Non-Functional Requirements with Aspects: An Industrial Case Study"

10.3.1 Description

The research article "Testing Non-Functional Requirements with Aspects: An Industrial Case Study" by Jani Metsä of Nokia in association with Mika Katara and Tommi Mikkonen [45] of Tampere University of Technology was published in 2007 as part of the Seventh International Conference on Quality Software. According to the authors, the main goal of software testing is to ensure that requirements are met by the implementation. There are already several systematic approaches for testing functional requirements, but only few for testing non-functional requirements. To tackle this problem, they turned to aspect-oriented programming as a potential testing technique. In their paper, they try to answer the following research questions:

1. To what extent can aspect-oriented techniques be harnessed to test non-functional requirements?

2. Which non-functional requirements lend themselves for testing with aspects?

The authors conducted a case study, analysing 150 requirements of an existing industrial embedded system (a quality verification software for mobile phones running Symbian OS). They were able to identify 16 crosscutting non-functional requirements for which seven test aspects were formulated and implemented. In detail, in the first step, a set of provided system requirements or characteristics are analysed and corresponding non-functional requirements (which might be crosscutting) derived. One non-functional requirement is derived from one or more system requirements, a system requirement can comprise multiple non-functional requirements. In a second step, the non-functional requirements are categorised (performance, robustness, ...), and corresponding testing objectives are derived. One testing objective is derived from one or more non-functional requirements. Finally, the test aspects can be formulated based on the testing objectives and non-functional requirements categories. One test aspect can comprise multiple testing objectives.

The approach proved to be easy and test coverage could be increased significantly. The use of aspects enables non-invasive testing throughout the software lifecycle. Furthermore, separation of (test) concerns can be achieved by modularizing crosscutting non-functional requirements. As a result, maintainability, reusability as well as tracing between requirements and test code can be improved. Based on the experiences gained, the authors concluded that especially system-wide and crosscutting non-functional requirements, such as security, performance, reliability and robustness, are suitable for aspect-oriented testing. To sum up, the article presents a systematic approach – a systematology - to derive testing objectives and test cases with related test aspects from non-functional requirements. It does not provide automation for any step, hence the main problems the authors identify: the lack of tool support and the need for testing personnel to be firm with aspect-oriented programming.

10.3.2 Application

The non-functional requirements (REQ) are derived from the user task or the corresponding system characteristics (Table 10.4). They have system-wide effects (hence affect multiple system-functions) and are of crosscutting nature. For example, REQ5 and REQ6 set performance constraints on REQ7 and REQ8 as well as REQ3; REQ3 and REQ4 set reliability and robustness constraints on each other; REQ9 sets security constraints on REQ1 and REQ2.

Subsequently, the non-functional requirements are categorised and corresponding test objectives can be derived (Table 10.5).

Finally, a test aspect for one or multiple requirement categories is formulated (Table 10.6). These test aspects would then need to be implemented manually using an AOP-Framework (AspectC++ [6], AspectJ [23], ...). For instance, the Memory Aspect could work like this: every time the constructor or destructor of an entity class (Movie, Performer, ...) is called, a counter will be incremented or decremented by the size of the class.

Table 10.4: Initial System Requirements of the Movie Manager App.

System Characteristic	**Derived Requirement**
Data is managed consistently by the system.	REQ1: Default values are provided (wherever possible). REQ2: Entities are linked consistently (a performer must always be linked to at least one movie).
The system is fault tolerant and able to report faulty behaviour.	REQ3: The system can recover from hang situations. REQ4: The system can identify correct and incorrect system behaviour.
The system runs on a mobile phone with limited amount of resources and thus has a strict memory footprint.	REQ5: The system must occupy at maximum W bytes of ROM REQ6: The system must occupy at maximum X bytes of RAM.
The system is fast and responsive.	REQ7: The system can respond to requests after Y time units after power-up. REQ8: All user requests are handled in Z time units.
The system might hold sensitive data and is therefore secure.	REQ9: The system follows the Android App security best practices (e.g. a password is required for sensitive data changes).

Table 10.5: Testing Objectives for Non-Functional Requirements.

REQ	**Testing Objective**	**Requirement Category**
REQ1 REQ2	TO1: Supervise data consistency and integrity.	Security (Integrity).
REQ9	TO2: Check password protection.	Security
REQ3	TO3: Generate hang situation.	Robustness
REQ3 REQ4	TO4: Analyse system reliability.	Reliability
REQ5 REQ6	TO5: Supervise memory consumption.	Performance (Memory)
REQ7	TO6: Measure time consumed from power-on to the system being in a responsive state.	Performance
REQ7 REQ8	TO7: Measure time consumed on serving requests and executing system-functions.	Performance

Table 10.6: Formulated Test Aspects.

Test Aspect	**Description**	**Test Objective(s)**
Integrity Aspect	Checks if all specified default values are provided and performers are linked to at least one movie.	TO1
Security Aspect	Tries to execute all data changing system-functions without providing the password. It should ‘t be possible to commit the changes.	TO1, TO2
Robustness Aspect	Generates a request jam to test if the SUT can recover from the hang situation.	TO3
Reliability Aspect	Collects information on SUT states and failures.	TO4
Memory Aspect	Supervise memory consumption by tracking all memory allocations and deallocations.	TO5
Performance Aspect	Measure function execution times.	TO6, TO7

10.4 Approach 2: "ACRE: An Automated Aspect Creator for Testing C++ Applications"

10.4.1 Description

Etienne Duclos, Sébastien Le Digabel, Yann-Gaël Guéhéneuc and Bram Adams [22] of École Polytechnique de Montréal published their article "ACRE: An Automated Aspect Creator for Testing C++ Applications" in 2013 as part of the 17th European Conference on Software Maintenance and Reengineering. They state that software should be faultless and in accordance with requirements yet testing costs need to be acceptable. Systematic and developer-friendly tools for testing functional and non-functional requirements are required to achieve this goal. However, such tools are sparce, especially for non-functional requirements. The authors review related approaches for testing with aspects, including the approach by Metsä et al., and conclude that high-level tool support and automation are required to solve the problems raised in these publications. To achieve this, they try to answer the following tripartite research question:

Can automatically generated test aspects be used to ...

1. ... detect memory leaks?

2. ... test invariants?

3. ... detect interference bugs?

The authors present ACRE (automated aspect creator), a tool that automatically generates test aspect code using a domain specific language. Provided a set of test cases or objectives or a bug report, the DSL statement describing the test aspect can be derived. The type of the bug or the category of the underlying NFR of the test case determines the type of the test aspect that must be chosen in this step. Afterwards, the test aspect is generated automatically based on the DSL description. ACRE takes the entire source code as an input and looks for DSL statements. They are parsed to generate the corresponding test aspects.

Using ACRE, the authors were able to detect one error of each type (1. – 3.) in the mathematical optimisation software NOMAD. Thanks to the DSL description of the test aspect, testing personnel does not need to have extensive knowledge about aspect-oriented programming, just about the DSL syntax. Although the creation of test cases from requirements or bug reports is not automated, the DSL and the types of available test aspects provide support in that regard. However, this means that the approach is currently limited to specific use cases (memory leaks, invariants, interference bugs in C++ applications). Nevertheless, it would be easy to extend the available functionalities and transfer the approach to other programming languages with support for aspect-oriented programming.

To sum up, the article presents a tool for the automatic generation of test aspects with given test cases formulated in a domain specific language.

10.4.2 Application

The non-functional requirements with corresponding test objectives or a bug report must be given, as the approach does not present a way to derive them. Let us consider the same requirements (REQ1-9) and test objectives (TO1-7) as in Table 10.4 and Table 10.5. Furthermore, a user of the Movie Manager App has filed the following bug report:

Table 10.7: Bug Report.

Summary: Memory Leak when removing all movies of a performer.
Description: Deleting all Movies linked to a performer leads to the removal of the performer from the performers list, but no memory* is not actually freed.
Steps to reproduce: 1. Consider a performer with one linked movie. 2. Delete the linked Movie.
Actual results: The performer and movie are removed from the respective lists, but no memory is freed.
Expected results: The performer and movie are removed from the respective lists and both objects are no longer in memory.

*(*For the purpose of this example, let us consider all data is kept in main memory.)*

Based on the given test objectives or the bug report, the test aspects must be formulated manually using a domain-specific language. However, the DSL provides certain types of Aspects (Counter, Logging, Timing, Checking) and guidelines that help formulate the test cases. Table 10.8 shows the aspect types corresponding to the given test objectives and test aspects of approach 1. The DSL formulation to test for memory leaks in the Performer class could look like this:

Listing 10.2: DSL Statement For Counter Aspect

```
1 // // name: MemoryAspect
2 // // type: Counter
3 // // className: Performer
4 // // namespace: MovieManager
```

The parameters *'name'* and *'type'* are required and determine the name and type of the test aspect. 'className' and 'namespace' are optional and specify, which class are concerned by the aspect. By default, the file in which the DSL statement was found is searched for namespaces and the name of the file is considered to be the name of the class. It must be placed somewhere within the source code, for example above or below the tested function or class or in a separate file. Afterwards, ACRE takes the entire source code as an input, looks for DSL statements (always starting with // //) and generates the corresponding test aspects automatically. For the example above, ACRE generates the memory aspect for the Performer class as shown in Listing 10.3. In essence, the counter aspect initialises a static counter variable that is incremented each time the Performer constructor is called and decremented each time the destructor is called. After the execution of the main method, the final count as well as a warning in case of a memory leak (counter > 0) are printed. In this manner, a class causing a memory leak can be detected. The timing and logging aspects are quite similar to the counter aspect. The checking aspect is more complex and allows the declaration of variables as well as the use of for-loops, while-loops and if-clauses. Note that the timing aspect in its current form is used for

interference bug testing, hence it measures access times to variables. However, a timing aspect for measuring function call times could easily be implemented by starting a timer before each function call and stopping it after each function call in the advice of the aspect.

Table 10.8: Aspect Types for Given Test Objectives.

Test Aspect	**Description**	**Test Objective(s)**	**Aspect Type**
Integrity Aspect	Checks if all specified default values are provided and performers are linked to at least one movie.	TO1	Checking
Security Aspect	Tries to execute all data changing system-functions without providing the password. It should ‘t be possible to commit the changes.	TO1, TO2	Checking
Robustness Aspect	Generates a request jam to test if the SUT can recover from the hang situation.	TO3	Checking, Logging
Reliability Aspect	Collects information on SUT states and failures.	TO4	Logging
Memory Aspect	Supervise memory consumption by tracking all memory allocations and deallocations.	TO5	Counting
Performance Aspect	Measure function execution times.	TO6, TO7	Timing (not in current form)

Listing 10.3: Generated Counter Aspect Code.

```
aspect MemoryAspect{
   public static int _Eval_Performer = 0;

   pointcut Eval_Performer() = "MovieManager::Performer";
   advice Eval_Performer() : slice struct{
      class Eval_Performer{
         public:
            // constructor -> increment
            Eval_Performer(){
               MemoryAspect::_EvalPerformer++;
            }
            // destructor -> decrement
            ~Eval_Performer(){
               MemoryAspect::_EvalPerformer--;
            }
      }
   };

   // print counter (and warning)
   advice execution (main(...)) : after () {
      printf("Final␣count␣of␣Eval_Performer:␣\%d\n",  _Eval_Performer)
      if(_Eval_Performer > 0)
         printf("Memory␣Leak!␣\n")
   }
};
```

10.5 Comparison

The approaches have been compared using a set of synthesis questions, as shown in Table 10.9.

Both approaches offer the same benefits: using test aspects, it is possible to test system-wide crosscutting non-functional requirements in a non-invasive way (without modifying or instrumentalising the source code). As a result, modularity, maintainability reusability as well as traceability of requirements to corresponding tests can be improved. Multiple stakeholders can benefit from both approaches, in particular testing and maintenance personnel. Correspondingly, both approaches use and produce similar artifacts: test objectives are derived from requirements and ultimately implemented using aspects.

However, there are differences concerning the specific individual steps, the level of automation as well as the relating quality of the approaches. Approach 1 describes a systematology to derive aspect test cases from requirements. These have to be implemented manually. In contrast, approach 2 assumes the test objectives to be provided and generates corresponding test aspects automatically. Subsequently, the most significant difference is the tool support: approach 1 does not provide any tool support, but rather describes a systematology. Approach 2 fully automates the final step of approach 1 – from test objectives to test code – and assists with the set-up of the concrete test cases through the structure of the domain specific language.

This is reflected in the evaluation of the respective approach as well. Both draw the conclusion that aspects are well suited for testing non-functional requirements. Approach 1 criticises the lack of tool support, approach 2 counteracts this exact problem. In Summary, the two approaches are quite similar. However, approach 1 is less extensive concerning the automation of the technique and should be regarded as a rather fundamental research. Approach 2 expands on the ideas of approach 1 and enhances their quality by partially automating the process.

Table 10.9: Synthesis Matrix.

No.	Approach 1: Testing Non-Functional Requirements with Aspects	Approach 2: ACRE
1a)	system requirements in natural language (provided), NFRs (derived), testing objectives (derived), test aspects (derived)	test objectives or bug report (provided), DSL-statements within the source code (derived), test aspects (generated)
1b)	system requirements	test objectives or bug report
1c)	Step 1: Provided a set of system requirements, NFRs (which might be crosscutting) are derived. One NFR is derived from one or more system requirements, a system requirement can comprise multiple NFRs. Step 2: The NFRs are categorised and corresponding testing objectives are derived. One testing objective is derived from one ore more NFRs. Step 3: Test aspects can be formulated based on the testing objectives and NFR categories. One test aspect can comprise multiple testing objectives.	Step 1: Provided a set of testing objectives or a bug report, the DSL statement describing the test aspect can be derived. The type of the bug or the category of the underlying NFR of the testing objective determines the type of the test aspect that must be chosen in this step. Step 2: The test aspect is generated automatically based on the DSL description.
2a)	Easy and non-invasive testing throughout the software lifecycle: during initial development for debugging or validating NFRs, during system testing and for maintaining tasks. Separation of concerns by modularizing crosscutting NFRs. Tracing of NFRs and corresponding tests	Easy and non-invasive testing throughout the software lifecycle: during initial development for debugging or validating NFRs, during system testing and for maintaining tasks. Separation of concerns by modularizing crosscutting NFRs. Tracing of NFRs and corresponding tests
2b)	Developers, testing and maintenance personnel	Developers, testing and maintenance personnel
2c)	Software Requirements (NFRs, Requirements Tracing), Software Testing (System Test), Software Maintenance	Software Requirements (NFRs, Requirements Tracing), Software Testing (System Test), Software Maintenance
3a)	none (except the aspects themselves)	ACRE (Automated Aspect Creator), DSL
3b)	none	Test cases and testing objectives must be derived manually from the requirements. The DSL facilitates the test case design. The test aspect code is generated automatically from the DSL statements.
4a)	Case study with requirements of an existing industrial embedded system. Comparison of test coverage with and without test aspects.	An empirical study aiming to find errors in the mathematical optimisation software NOMAD was conducted. The approach was contrasted with other common techniques and tools.
4b)	Test aspects proved to be an easy and non-invasive technique for testing NFRs. Separation of (test) concerns by modularising NFRs. BUT: lack of tool support, complicated build process.	Automatically generated test aspects proved to be suitable for finding memory leaks, interference bugs and testing invariants. The approach is easy to use because developers do not need to know aspect-oriented programming itself, only the DSL syntax and semantics. BUT: derivation of test cases from requirements and DSL statement input still manual

10.6 Conclusion

In summary, this report presents two articles describing possible approaches to harness aspect-oriented techniques for testing non-functional requirements. A literature search based on the first article was carried out to obtain an overview of the current state of research and find relevant literature for this report. The selected article builds upon the ideas of the given article and presents a tool for (partially) automating the creation of test aspects. The two approaches have been compared and illustrated using a literature synthesis with a set of synthesis questions and a synthesis matrix as well as an application to an example.

The results of both research articles and the synthesis and example, conducted for the report, indicate that aspects are suitable for testing non-functional requirements in a systematic manner due to multiple reasons: Firstly, the basic functionality of aspects carries an inherent systematology and lends itself for testing. An aspect comprises a set of statements that can be executed every time before, after or around a function call, which can easily be utilised to check pre- and postconditions or to measure resource consumption and runtimes. In addition to that, they are non-invasive, meaning the tested source code does not need to be modified and the test aspect can easily be separated from the main code, reused or removed. Furthermore, test aspects are especially suitable for testing non-functional requirements because they modularise the system level concerns, non-functional requirements pose, thus maintaining separation of concerns. System-wide functionality, that is spread out through the entire system and its source code can be tested using one aspect. This also improves traceability between the non-functional requirements and the corresponding tests.

Metsä et al. point out that the lacking tool support and the need for developers to have a firm understanding of aspect-oriented programming pose possible challenges. Duclos et al. solve these problems (partially) by introducing ACRE, a tool that parses DSL statements to generate test aspects automatically. However, the tool only supports four aspect types and is limited to C++ applications. The lack of subsequent research articles (after 2013) indicates that systematic testing using aspects is not in focus of current research anymore. Nevertheless, existing approaches and tools, as presented in this report, should be further refined to make use of the aforementioned advantages of aspect-oriented testing, especially for non-functional requirements. Because aspect-oriented programming lost popularity in recent years, applying aspect-oriented testing techniques in practice is difficult because only few developers and testers are able to implement them. However, using a domain specific language as an intermediate layer of abstraction, as described by Duclos et al, tackles this problem and enables relevant stakeholders to use test aspects as small, re-usable and none-invasive test modules throughout the software lifecycle at the system test level.

11 Conclusion

This chapter presents first of all the most important insights from each of the individual chapters. For this, the conclusions from each topic are summarized. Then, the improved knowledge areas from the SWEBOK are listed and their use over the chapters is compared. In the end, a final conclusion over all the topics is presented.

Chapter 2 presents different approaches to create acceptance tests that can be automatically executed with the tool FitNesse. Due to the few results in the literature search, it seems that this topic using the specific tool FitNesse is not popular in the research. The first presented approach [24] is aimed at larger projects and uses lots of artifacts in the process. The second presented approach [42] is aimed to smaller projects and is adjusted for the use of US-UIDs, a type of artifact that was invented by some of the authors of the article. Whilst the first approach should be executed by a business analyst, the second approach is designed to be executed by the customer and the developers combined. Both presented approaches are highly dependent on the experience and skill of the person executing the approach. Most of the steps are done manually and require human judgement. Therefore, the approaches are only recommended if the required artifacts are already part of the engineering process and a person with experience with these artifacts takes part in the engineering process.

Chapter 3 showcases approaches to automatically derive test scenarios from means of the specification area using transition systems. This improves traceability between the specification and implementation and allows for an easy validation of requirements. The test cases can be derived by traversing the transition system. The first approach [47] focuses on system level test generation whilst the second presented approach [50] focuses on use case level test generation. Both presented approaches do not generate a sufficient amount of robustness tests for fault detection and do not sufficiently cover data variations. However, the presented approaches are still recommended for the automatic generation of functional test scenarios.

Chapter 4 presents approaches to test real-time requirements. The literature search showed that this topic is not popular in the research. One of the presented approaches [53] involves manually executed, intermediate steps. Therefore, this approach cannot be recommended in the presented form. The other approach [30] executes all intermediate steps automatically and can be recommended for usage.

Chapter 5 focuses on testing with a classification tree. This technique offers the possibility to reduce the number of test cases. The first presented approach [**?**] describes how requirements can be transformed into a classification tree for the input parameters of a system model. Another presented article [**?**] shows that classification trees are, despite existing, more recent methods, still relevant.

Chapter 6 is missing.

In **Chapter 7** the topic *testing with system models* is discussed. The literature search showed that the articles relevant to this topic are mostly published between 2000-2009. Both presented approaches use requirements traceability in their model-based test generation processes and provide a tool support to generate tests automatically. The first approach [2] provides test generation tool Qtronic, which is not existing anymore. Therefore, the second approach [12] could be recommended for the automatic test generation with system models, because it provides LEIRIOS Test Designer tool (currently named Smartesting), which is still available. However, both approaches lack the detailed explanation on how the test cases are generated via the provided test generation tools.

Chapter 8 presents approaches to automatically generate test cases for functional and non-functional requirements from User Requirements Notation. The literature search showed that this topic has not been researched enough, as this test generation process requires an intermediate step. The first presented approach [4] addresses both functional and non-functional requirements, however, the provided Validation Framework is not available and without it, this approach cannot be used. Although the second presented approach [13] generates test cases only for functional requirements, it is still recommendable as the required frameworks, jUCMNav and JUnit, are still available and the whole process can be done quickly without problems.

Chapter 9 focuses on testing non-functional requirements with risk analysis. Both presented approaches deal with non-functional requirements and mention risk analysis only as a side note. Although the first presented approach [61] describes how non-functional requirements can be defined and controlled, it does not specify a way to test them except for simulating the operating condition. The second presented approach [39] leaves it to software architects to define non-functional requirements and uses only architectural non-functional requirements that exist as code conventions and other guidelines. The conclusion is that non-functional requirements with higher risks need to be paid more attention to by giving the tests higher priority.

Chapter 10 presents approaches to harness aspect-oriented techniques for testing non-functional requirements. While the first presented approach [45] points out the lacking tool support and the need for developers to have a firm understanding of aspect-oriented programming pose possible challenges, the second presented approach [22] solves these problems (partially) by introducing ACRE. ACRE is a tool that parses DSL statements to generate test aspects automatically, which only supports four aspect types and is limited to C++ applications. The literature search showed that systematic testing using aspects is not in focus of current research anymore due to its implementation difficulty. Nevertheless, existing approaches and tools should be further refined to make use of the advantages of aspect-oriented testing, especially for non-functional requirements.

Table 11.1 presents the improved knowledge areas of the SWEBOK according to the synthesis matrices of the individual chapters. As shown in Table 11.1, knowledge areas from the field of *Software Testing* were improved by every topic. Improvements in the field of *Software Requirements* are part of every topic except the topic presented in chapter 5. The topics presented in chapter 5, 8, 9 and 10 specifically improved a knowledge area from the field of *Software Maintenance*. The field of *Software Engineering Models and Methods* is improved by the topics presented in chapter 3, 4, 7 and 8. Only the approaches from the topic of chapter 2 and one of the approaches from the topic of chapter 5 are mentioned to help with *Software Construction*.

Table 11.1: Improved knowledge areas of the SWEBOK. The rows represent the knowledge areas and the columns represent the topics. If an area is improved by a topic, the entry is marked with a *X*.

	2	3	4	5	7	8	9	10
Software Testing	X	X	X	X	X	X	X	X
Software Requirements	X	X	X		X	X	X	X
Software Maintenance				X		X	X	X
Software Engineering Models and Methods		X	X		X	X		
Software Construction	X			X				

Furthermore, while the topics presented in chapter 7 and 10 are not popular in current research, the topics presented in chapter 2, 4 and 8 are generally not popular in research. However, still two different approaches could be found and are described for each topic of this work. Most of the topics of this work turned out to have potential to be automated at least partly. According to the synthesis matrices of the individual chapters other than the topics presented in chapter 5 and 9, every topic included at least one approach that uses some automation steps for the creation of tests. Still, most of the work has to be done manually. Only the topic presented in chapter 3 includes an approach that is mostly automated.

In Summary, most of the presented approaches are recommended under appropriate circumstances. These circumstances include having persons with the needed skills as in chapter 2 or having access to specific tools as in chapter 7 and 8. It should be mentioned that even though these approaches seem useful, often certain steps are not explained in detail. In particular, this is the case for the test generation steps such as in chapter 3, 7, and 9.

Glossary

Acceptance Test Test to validate that a software meets its requirements by simulating how end-users typically conduct business using the system. Identifying and designing acceptance tests may be difficult for nonfunctional requirements. To be validated, they must first be analyzed and decomposed to the point where they can be expressed quantitatively [16].

Acceptance Test Driven Development (ATDD) Use of automated acceptance tests with the additional constraint that these tests are written before the application code get implemented [42].

All-Interface Coverage Criterion (AIC) Test criterion for component based systems, which ensures that each interface is tested once [30].

All-Interface-Event Coverage Criterion (AIEC) Test criterion for component based systems, which combines edge coverage with logical coverage (CACC). Edge coverage determines whether each edge in the control flow graph has been executed. Decisions in the event handler are modeled as predicate expressions, where each clause is an event. CACC is used to create the essential test paths from this [30].

All-Interface-Transition Coverage Criterion (AITC) Test criterion for component based systems, which ensures that each interface between components is tested once and each internal state transition path in a component is toured at least once [30].

Anti-Lock Braking System (ABS) Detects the tendency of one or more wheels to lock at an early stage during braking and then immediately ensures that the brake pressure is kept constant or reduced. In this way, the wheels do not lock and the vehicle follows the steering. This allows a car to be braked or brought to a standstill safely and quickly [32].

Aspect System-wide functionality or issues isolated from the main business logic of the program [37].

Aspect Oriented Programming (AOP) Programming paradigm using aspects to encapsulate crosscutting concerns. New functionality can be added to existing code without modifying it. Therefore, modularity, maintainability and reusability can be improved [37].

AspectC++ Aspect-oriented programming extension for the C and C++ programming languages [6].

AspectJ Aspect-oriented programming extension for the Java programming language [23].

Classification Tree (CT) According to CT-method, the input domain of a test object is analyzed on the basis of its functional specification with respect to various aspects regarded as relevant for the test. For each aspect, disjoint and complete classifications are formed.

Classes resulting from these classifications may be further classified iteratively. The stepwise partition of the input domain by means of classifications is represented graphically as a tree. Subsequently, test scenarios are formed by combining classes of different classifications [20].

Component Interface Interaction Graph (CIIG) A special CREMTEG, which represents the time-dependent connectivity relationships between these components as well as time-dependent relations inside a component and a component interface [30].

Component State-Based Event-Driven Interaction Behavior Graph (CSIEDBG) A special CREMTEG, which, beside CIIG and CSIBG, deals with concurrency. Therefore, the CSIBG model is extended as follows: First, each provider service in a component has one event capture to process the incoming external message. Second, each state contains one event handler and each event capture generates one event in one state. Last, all events are handled by one event handler [30].

Component State-Based Interaction Behavior Graph (CSIBG) A special CREMTEG, which additionally to a CIIG reflects the internal behavior of a component by state transitions and internal message edges with time stamps [30].

Component-Based Real-Time Embedded Model-Based Test Graph (CREMTEG) The CREMTEG models are derived from component level sequence diagrams and component state diagrams involved in the component interactions. Test criteria for generating integration tests are developed from the CREMTEG models [30].

Contract A requirement-level logical expression used to express preconditions and postconditions of an operation [47].

Correlated Active Clause Coverage (CACC) A special form of predicate logic. For each predicate p and each major clause c_i in C_p (set of all clauses in p), choose minor clauses c_j, $j \neq i$, so that c_i determines p. c_i has two requirements: c_i evaluates to true and c_i evaluates to false. The values chosen for the minor clauses c_j must cause p to be true for one value of the major clause c_i and false for the other, that is, it is required that $\mathrm{p}(c_i = \text{true}) \neq \mathrm{p}(c_i = \text{false})$ [7].

Coverage Criterion A criterion used to limit the amount of test objectives to be generated. It specifies which paths to traverse in the context of the traversal of a transition system [47].

Crosscutting Concerns Part of a software that affects other parts, impeding the separation of concerns. This can lead to concern scattering (duplications) or tangling (dependencies) [45].

Domain Specific Language (DSL) Small language, focused on a particular aspect of a software system [28].

Event Sequence Graph (ESG) Used to represent system behavior as well as user-system interaction by events. An ESG is a more abstract representation compared to a state transition diagram of a finite-state automaton [10].

Extendend Automation Method (EXAM) Test method used by the AUDI AG and within the Volkswagen AG to perform tests at component and system levels. EXAM defines the process, the roles, and the tools used to model test cases graphically in UML. Test automation

in the scope of EXAM comprises the automated generation of platform dependent code and the automated execution of the derived test suite without human interactions [53].

Functional Requirements (FRs) Describe the functions that the software is to execute, e.g. formatting some text or modulating a signal. They are sometimes known as capabilities or features. A FR can also be described as one for which a finite set of test steps can be written to validate its behavior [16].

Goal-Oriented Requirements Language (GRL) A visual modeling notation for intentions, business goals, and NFRs of many stakeholders, for alternatives that have to be considered, for decisions that were made, and for rationales that helped make these decisions [8].

Implementation Under Test (IUT) The part of a real system which is to be tested, which should be an implementation of applications, services or protocols [25].

Interaction Overview Diagram (IOD) A special form of activity diagram used to show control flow. Each node in the IOD represents either an interaction diagram (sequence diagrams) or interaction occurrences that show an operation invocation [50].

JUnit Testing Framework A simple framework to write repeatable tests. It is an instance of the xUnit architecture for unit testing frameworks [36].

Model Based Black-Box Testing (MBBBT) Approach to define test scenarios for software developed in a model-based way from two different perspectives and to create consistency between both requirement-based test design and model-based test design [20].

Model-Based Testing (MBT) Software testing technique using a test model to automatically derive test cases following different coverage criteria and therefore shortening the time required for testing [2].

Non-Functional Requirements (NFRs) Act to constrain the solution, sometimes known as constraints or quality requirements. They can be further classified according to whether they are performance requirements, maintainability requirements, safety requirements, reliability requirements, security requirements, interoperability requirements or one of many other types of software requirements [16].

Object Constraint Language (OCL) A formal language used to describe expressions on UML models. These expressions typically specify invariant conditions that must hold for the system being modeled or queries over objects described in a model. Therefore, OCL is often used to define contracts [48].

Operation An abstract term used to describe a state transition in a transition system. An operation is equal to a use case or to an interaction diagram/interaction occurrence for instance [47, 50].

Separation Of Concerns Programming principle aiming to split the main task of a program into multiple sub tasks and solving them individually [45].

Stakeholders In a software development process, the software systems are built, tested, maintained, enhanced, and paid. All these activities involve a number of people in building the software. Each of these activities has a different group of users working on it, which may have different interests, requirements for making the software. All these different groups of people comprise stakeholders. Therefore, we can define a stakeholder as an architect of an organization, team, or group having an interest in making a product [54].

System Model (SM) An abstract model primarily created for system development and then used for testing as well [2].

Systems Modeling Language (SysML) A general-purpose architecture modeling language for Systems Engineering applications. It supports the specification, analysis, design, verification and validation of a broad range of systems and systems-of-systems. SysML is a dialect of UML 2, and is defined as a UML 2 profile [56].

Test Case A test case is a documented set of preconditions (prerequisites), procedures (inputs/actions), and postconditions (expected results) which a tester uses to determine whether a SUT satisfies use case requirements or works correctly. A test case can have one or multiple test scripts and a collection of test cases is called a test suite [55].

Test Model (TM) A behavioral model of the system solely developed for testing [2].

Test Objective Purpose of a test. Stating the objectives of testing in precise, quantitative terms supports measurement and control of the test process. A test objective is used in [47] as a synonym for test path as a combination of abstract operations to be composed into one test scenario [16, 47].

Test Scenario A Test Scenario is a statement describing the functionality of the application to be tested. It is used for end to end testing of a feature and is generally derived from the use cases. Test scenarios can serve as the basis for lower-level test case creation. A single test scenario can cover one or more test cases. Therefore a test scenario has a one-to-many relationship with the test cases [5].

Timed Usage Model (TUM) Markov Chain Usage Models (MCUM) extended by time information, that preserve the semantic of the MCUM and support automated test case generation for embedded systems in test environments as they are established in the automotive industry [53].

Transition System Used to derive test scenarios by helping to generate test objectives through its traversal. Consists of states and transitions, where states are given by contracts and transitions are given by operations [47].

UC-SCSystem A prototype-tool using use case scenarios to derive executable test scenarios as JUnit tests [47].

UC-System A prototype/interpreter-tool used to build a transition system and to derive test objectives from it [47].

Unified Modeling Language (UML) A standard visual modeling language intended to be used for modeling business and similar processes as well as the analysis, design, and implementation of software-based systems [59].

Use Case A type of scenario that provides and embodies a context during the elicitation of user requirements. Is used in [47] as an abstract, requirement-level term used to describe a main functionality of a system [16, 47].

Use Case Map (UCM) The UCM visual scenario notation focuses on the causal flow of behavior optionally superimposed on a structure of components. UCM depict the causal interaction of architectural entities while abstracting from message and data details [8].

Use Case Scenario A synonym for sequence diagram in case of [47].

User Requirements Notation (URN) A modeling language that aims to support the elicitation, analysis, specification, and validation of requirements. URN is the first international standard to address explicitly, in a graphical way and in one unified language, goals and scenarios, and the links between them. URN models can be used to specify and analyze various types of reactive systems as well as telecommunications standards and business processes. URN allows software and requirements engineers as well as business analysts to discover and specify requirements for a proposed system or process (or evolving ones), and analyze such requirements for correctness and completeness. The URN standard combines two sub-languages: GRL for modeling actors and their intentions, and the UCM notation for describing scenarios and architectures [8].

XML Metadata Interchange (XMI) XML based metadata interchange format gained from an Interaction Overview Diagram to generate a transition system [50].

12 Bibliography

[1] Abbors F., Bäcklund A., Truscan D.: "MATERA - An Integrated Framework for Model-Based Testing," 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems, Oxford, 2010, pp. 321-328, doi: 10.1109/ECBS.2010.46.

[2] Abbors F., Truscan D., Lilius J.: "Tracing Requirements in a Model-Based Testing Approach," 2009 First International Conference on Advances in System Testing and Validation Lifecycle, Porto, 2009, pp. 123-128, doi: 10.1109/VALID.2009.15.

[3] ACM Digital Library. https://dl.acm.org/

[4] Arnold, D., Corriveau, J.-P., and Shi, W.: "Scenario-Based Validation: Beyond the User Requirements Notation," 2010 21st Australian Software Engineering Conference, Auckland, 2010, pp. 75-84, doi: 10.1109/ASWEC.2010.29.

[5] Art Of Testing: Test Scenario, https://artoftesting.com/test-scenario-examples, last accessed 2021/02/01.

[6] AspectC++ Development Team: About the Project. https://www.aspectc.org/, last accessed 2020/12/28.

[7] Ammann, P., Offutt, J.: "Introduction to Softwate Testing Chapter 3.1, 3.2 Logic Coverage," https://www.cs.montana.edu/courses/se422/currentLectures/Ch3-1-2.pdf, last accessed 2021/01/28.

[8] Amyot, D., Mussbacher, G.: "User Requirements Notation: The First Ten Years, The Next Ten Years," Journal of Software, vol. 6, no. 5, pp. 747-768, 2011, doi:10.4304/jsw.6.5.747-768.

[9] Barmi, Z. A., Ebrahimi, A. H., Feldt, R.: "Alignment of Requirements Specification and Testing: A Systematic Mapping Study," 2011 IEEE Fourth International Conference on Software Testing, Verification and Validation Workshops, Berlin, 2011, pp. 476-485, doi: 10.1109/ICSTW.2011.58.

[10] Belli, F., Hollmann, A.: "A Graph-Model-Based Testing Method Compared with the Classification Tree Method for Test Case Generation," 2009 Third IEEE International Conference on Secure Software Integration and Reliability Improvement, Shanghai, 2009, pp. 193-200, doi: 10.1109/SSIRI.2009.40.

[11] Bernard, E., Bouquet, F., Charbonnier, A., Legeard, B., Peureux, F., Utting, M., Torreborre, E.: 2006. "Model-based testing from UML models," In: Hochberger, C. & Liskowsky, R. (Hrsg.), INFORMATIK 2006 – Informatik für Menschen – Band 2, Beiträge der 36. Jahrestagung der Gesellschaft für Informatik e.V. (GI). Bonn: Gesellschaft für Informatik e.V. (S. 223-230), ISSN: 1617-5468.

[12] Bernard E., Legeard B.: "Requirements Traceability in the Model-Based Testing Process," in Software Engineering, ser. Lecture Notes in Informatics, vol. 106. Bttinger, Stefan and Theuvsen, Ludwig and Rank, Susanne and Morgenstern, Marlies, 2007, pp. 45–54, ISSN: 1617-5468

[13] Boucher, M., Mussbacher, G.: "Transforming Workflow Models into Automated End-to-End Acceptance Test Cases," 2017 IEEE/ACM 9th International Workshop on Modelling in Software Engineering (MiSE), Buenos Aires, 2017, pp. 68-74, doi: 10.1109/MiSE.2017.5.

[14] Bouquet F., Grandpierre C., Legeard B., Peureux F., Vacelet N., Utting M.: 2007. "A subset of precise UML for model-based testing," In Proceedings of the 3rd international workshop on Advances in model-based testing (A-MOST '07). Association for Computing Machinery, New York, NY, USA, 95–104. doi: https://doi.org/10.1145/1291535.1291545

[15] Bouquet F., Jaffuel E., Legeard B., Peureux F., Utting M.: 2005. "Requirements traceability in automated test generation: application to smart card software validation," SIGSOFT Softw. Eng. Notes 30, 4 (July 2005), 1–7. doi: https://doi.org/10.1145/1082983.1083282

[16] Bourque, P., Fairley, R. E.: Guide to the Software Engineering Body of Knowledge. Version 3.0. IEEE Computer Society, 2014. https://cs.fit.edu/~kgallagher/Schtick/Serious/SWEBOKv3.pdf, last accessed 2021/02/06.

[17] Chair of Software Engineering of University Heidelberg. Guidelines for literature research. https://confluence-se.ifi.uni-heidelberg.de/x/XQAyDg, last accessed 2021/02/06.

[18] Chair of Software Engineering of University Heidelberg. Movie Manager Example. https://confluence-se.ifi.uni-heidelberg.de/x/IgC6Dg, last accessed 2021/02/06.

[19] Chen, L., Li, Q.: "Automated test case generation from use case: A model based approach," 2010 3rd International Conference on Computer Science and Information Technology, Chengdu, 2010, pp. 372-377, doi: 10.1109/ICCSIT.2010.5563772.

[20] Conrad, M., Fey, I., Sadeghipour, S.: "Systematic Model-Based Testing of Embedded Automotive Software," In: Electronic Notes in Theoretical Computer Science 111, 2005. https://doi.org/10.1016/j.entcs.2004.12.005

[21] Dijkstra, E.: "A Discipline of Programming," In: Prentice-Hall Series in Automatic Computation Englewood Cliffs, New Jersey, 1976.

[22] Duclos, E., Le Digabel, S., Guéhéneuc, Y., Adams, B.: "ACRE: An Automated Aspect Creator for Testing C++ Applications," 2013 17th European Conference on Software Maintenance and Reengineering, Genova, 2013, pp. 121-130, doi: 10.1109/CSMR.2013.22.

[23] Eclipse Foundation: AspectJ. https://www.eclipse.org/aspectj/, last accessed 2020/12/28.

[24] El-Attar, M., Miller, J.: Developing comprehensive acceptance tests from use cases and robustness diagrams. Requirements Eng 15, 2010, pp. 285–306, https://doi.org/10.1007/s00766-009-0088-6

[25] ETSI: Testing, Interoperability and Technical Quality, https://portal.etsi.org/CTI/CTISupport/Glossary.htm, last accessed 2021/02/01.

[26] FitNesse. http://docs.fitnesse.org/FrontPage, last accessed 2021/02/06.

[27] FitNesse: Writing Fit Tables. http://fitnesse.org/FitNesse.UserGuide.WritingAcceptanceTests.FitFramework.WritingFitTables, last accessed 2021/02/06.

[28] Fowler, M., Parsons, R.: "Domain Specific Languages Addison-Wesley Professional," 2010.

[29] Gregoriades, A., Sutcliffe, A.: "Scenario-based assessment of nonfunctional requirements," in IEEE Transactions on Software Engineering, vol. 31, no. 5, pp. 392-409, May 2005, doi: 10.1109/TSE.2005.59.

[30] Guan, J., Offutt, J.: "A model-based testing technique for component-based real-time embedded systems," 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Graz, 2015, pp. 1-10, doi: 10.1109/ICSTW.2015.7107407.

[31] Hausberger, F.: Research planning, research and mid-term presentation, https://github.com/fidsusj/SWE-Seminar, last accessed 2021/01/10.

[32] Heinz, J., Koch, D.: Antiblockiersystem ABS. In: Reif K. (eds) "Fahrstabilisierungssysteme und Fahrerassistenzsysteme (2010)," pp. 34-49. Vieweg+Teubner. https://doi.org/10.1007/978-3-8348-9717-6_3

[33] Ibrahim, R., Saringat, M. Z., Ibrahim, N., Ismail, N.: "An Automatic Tool for Generating Test Cases from the System's Requirements," 7th IEEE International Conference on Computer and Information Technology (CIT 2007), Aizu-Wakamatsu, Fukushima, 2007, pp. 861-866, doi: 10.1109/CIT.2007.116.

[34] IEEE Xplore. https://ieeexplore.ieee.org/Xplore/home.jsp, last accessed 2021/02/06.

[35] IMDB. https://www.imdb.com/, last accessed 2021/02/06.

[36] JUnit: JUnit, https://junit.org/junit4/, last accessed 2021/02/01.

[37] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J. M., Irwin, J.: (1997) "Aspect-oriented programming," In: Akşit M., Matsuoka S. (eds) ECOOP'97 — Object-Oriented Programming. ECOOP 1997. Lecture Notes in Computer Science, vol 1241. Springer, Berlin, Heidelberg. https://doi.org/10.1007/BFb0053381

[38] Kim, Y. G., Hong, H. S., Cho, S. M., Bae, D. H., Cha, S. D.: "Test cases generation from UML state diagrams," In: IEEE Proceedings - Software, vol. 146, no. 4, pp. 187-192, Aug. 1999. https://doi.org/10.1049/ip-sen:19990602.

[39] Lagerstedt, R.: "Using automated tests for communicating and verifying non-functional requirements," 2014 IEEE 1st International Workshop on Requirements Engineering and Testing (RET), Karlskrona, 2014, pp. 26-28, doi: 10.1109/RET.2014.6908675.

[40] Li, L., Miao, H.: "An Approach to Modeling and Testing Web Applications Based on Use Cases," 2008 International Symposium on Information Science and Engineering, Shanghai, 2008, pp. 506-510, doi: 10.1109/ISISE.2008.265.

[41] Longo, D., Vilain, P.: "Creating User Scenarios through User Interaction Diagrams by Non-Technical Customers," In SEKE, 2015, pp. 330-335, doi: 10.18293/SEKE2015-179.

[42] Longo, D., Vilain, P.: Pereira da Silva, L., Mello, R.: 2016. A web framework for test automation: user scenarios through user interaction diagrams. In Proceedings of the 18th International Conference on Information Integration and Web-based Applications and Services (iiWAS '16). Association for Computing Machinery, New York, NY, USA, 458–467. doi: https://doi.org/10.1145/3011141.3011158

[43] Malik Q. A., et al.: "Model-Based Testing Using System vs. Test Models - What Is the Difference?," 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems, Oxford, 2010, pp. 291-299, doi: 10.1109/ECBS.2010.41.

[44] Malik Q. A., Truscan D., Lilius J.: "Using UML Models and Formal Verification in Model-Based Testing," 2010 17th IEEE International Conference and Workshops on Engineering of Computer Based Systems, Oxford, 2010, pp. 50-56, doi: 10.1109/ECBS.2010.13.

[45] Metsä, J., Katara, M., Mikkonen, T.: "Testing Non-Functional Requirements with Aspects: An Industrial Case Study," Seventh International Conference on Quality Software (QSIC 2007), Portland, OR, 2007, pp. 5-14, doi: 10.1109/QSIC.2007.4385475.

[46] Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.-M.: 2003. "Requirements by Contracts allow Automated System Testing," In Proceedings of the 14th International Symposium on Software Reliability Engineering (ISSRE '03). IEEE Computer Society, USA, 85. https://dl.acm.org/doi/10.5555/951952.952350.

[47] Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.-M.: "Automatic test generation: a use case driven approach," in IEEE Transactions on Software Engineering, vol. 32, no. 3, pp. 140-155, March 2006, doi: 10.1109/TSE.2006.22.

[48] Object Management Group: Object Constraint Language, https://www.omg.org/spec/OCL/2.4/PDF, last accessed 2021/02/01.

[49] Parnas, D.: "On the criteria to be used in decomposing systems into modules," In: Communications of the ACM, 1972.

[50] Raza, N., Nadeem, A., Iqbal, M. Z. Z.: "An Automated Approach to System Testing Based on Scenarios and Operations Contracts," Seventh International Conference on Quality Software (QSIC 2007), Portland, OR, 2007, pp. 256-261, doi: 10.1109/QSIC.2007.4385504

[51] Sarma, M., Mall, R.: "System Testing using UML Models," 16th Asian Test Symposium (ATS 2007), Beijing, 2007, pp. 155-158, doi: 10.1109/ATS.2007.102.

[52] Science Direct. https://www.sciencedirect.com/, last accessed 2021/02/06.

[53] Siegl, S., Hielscher K.-S., German R.: "Model Based Requirements Analysis and Testing of Automotive Systems with Timed Usage Models," 2010 18th IEEE International Requirements Engineering Conference, Sydney, NSW, 2010, pp. 345-350, doi: 10.1109/RE.2010.49.

[54] Software Testing Class: Involvement Of Stakeholders In Testing, https://www.softwaretestingclass.com/involvement-of-stakeholders-in-testing/, last accessed 2021/02/01.

[55] Software Testing Fundamentals: Test Case, https://softwaretestingfundamentals.com/test-case/, last accessed 2021/02/01.

[56] SysML: SysML Open Source Project - What is SysML? Who created SysML?, https://sysml.org/, last accessed 2021/02/01.

[57] Springer Link. https://link.springer.com/, last accessed 2021/02/06.

[58] t2informatik GmbH. Requirement Diagram. https://t2informatik.de/en/smartpedia/requirement-diagram/, last accessed 2021/02/06.

[59] UML Diagrams: The Unified Modeling Language, https://www.uml-diagrams.org/, last accessed 2021/02/01.

[60] Zhang, X., Tanno, H.: "Requirements document based test scenario generation for web application scenario testing," 2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW), Graz, 2015, pp. 1-3, doi: 10.1109/ICSTW.2015.7107465.

[61] Zou, J., Pavlovski C. J.: "Control Cases during the Software Development Life-Cycle," 2008 IEEE Congress on Services - Part I, Honolulu, HI, 2008, pp. 337-344, doi: 10.1109/SERVICES-1.2008.46.

List of Figures

List of Tables