Author 1, Author n

Title

Technical Report

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

mm dd, year

Abstract

Place the abstract in this section.

Contents

1 Introduction

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

1.1 Subsection

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

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

- Aliquam
- mus
- montes

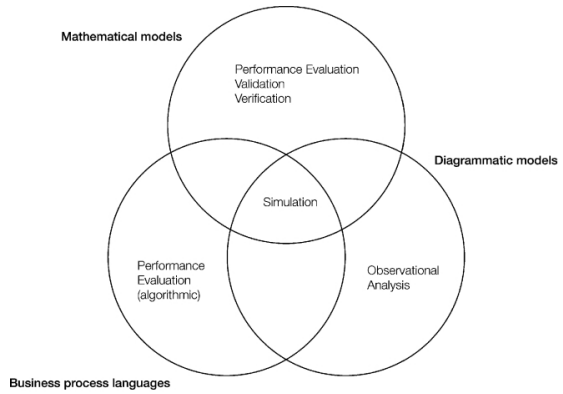

Figure 1.1: caption text

Table 1.1: Table caption text

X	Y
Item1	description
Item2	description
Item2	description

Subsubsection

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

2 Richtiges Zitieren

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

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

beachtet und eingehalten werden.

3 Chapter

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

4 Testing with a Transition System

4.1 Introduction

System tests are used to make sure that clients receive exactly the kind of software that they previously specified within the order contract submitted to the software vendor. Oftentimes what was specified and what was implemented in the end does not match entirely. The software vendor therefore faces traceability problems to track which requirements could be covered in which tests. 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 bé 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 through possible combinations of fine-grained functional requirements. 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 [3] and [4] were analyzed. While [3] was given in advance by the advisors, [4] was discovered through an extensive literature search shown in section 4.2. Both apporaches will be explained in the following sections 4.3 and 4.4 and applied to the movie management software example. A comparison between the two approaches will be drawn using a synthesis matrix shown in section 4.5. The main results of testing with transitions systems will be summarized in section 4.6.

Please refer to the glossar in order to receive a common understanding of the following terms used in the sections below: contract, operation, test criterion, test objective, test scenario, use case, use case scenario.

4.2 Literature Search

The literature research was driven by the central research question: „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 [3]. But as [3] 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 and ACM. Only some papers on ACM could not be accessed publicly. The

amount of results was considered to be sufficient to cover all relevant scientific papers, which is why research was restricted on these two platforms. Furthermore two relevance criteria inspired by the central research question 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).

Search was done using both snowballing and search term techniques. 140 papers were found referencing [3] and 46 articles were referenced by [3]. The results of the snowballing search can be found in Table 4.1. 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 basis knowledge to realize the transition system approach. Additionally, most of the references where quite old since the original paper was published in 2006. Therefore not all papers could be found on IEEE Xplore or ACM. Why specific papers were considered not suitable or only partly suitable is documented in [5].

Table 4.1: 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 havint the search term "test" and "use case" in its publication title were evaluated.

Table 4.2: Results of search-term based technique

Source	**Date**	**Search restrictions**	**Search query**	**#Results**
IEEE Xplore	2020-22-21	"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	2020-22-21	"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	2020-22-21	"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	2020-22-21	"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	2020-22-21	"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 cases" AND contracts	0
ACM	2020-22-21	"system tests" in title; "transition system" in full text; "use cases" in title	"system tests" AND "transition system" AND "use cases"	0
ACM	2020-22-21	"test" in title; "use case" in title	"system tests" AND "use cases"	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,
- An Automatic Tool for Generating Test Cases from the System's Requirements,
- Automated Test Case Generation from Use Case: A Model Based Approach,
- Requirements Document Based Test Scenario Generation for Web Application Scenario Testing,
- An Approach to Modeling and Testing Web Applications Based on Use Cases,

- Test cases generation from UML state diagrams,
- Requirements by Contracts allow Automated System Testing,
- An Automated Approach to System Testing Based on Scenarios and Operations Contracts.

The decision criteria are based on the different search terms above mentioned 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 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 validate test objectives. Further information on the paper will be introduced in the mid-term presentation.

4.3 Approach 1

4.3.1 Description of Approach 1

The approach consists of two phases (see Figure 4.1). 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 by writing them in the form of requirement-level first order logical expressions. They consist of predicates with logical operators, quantifiers and implications 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

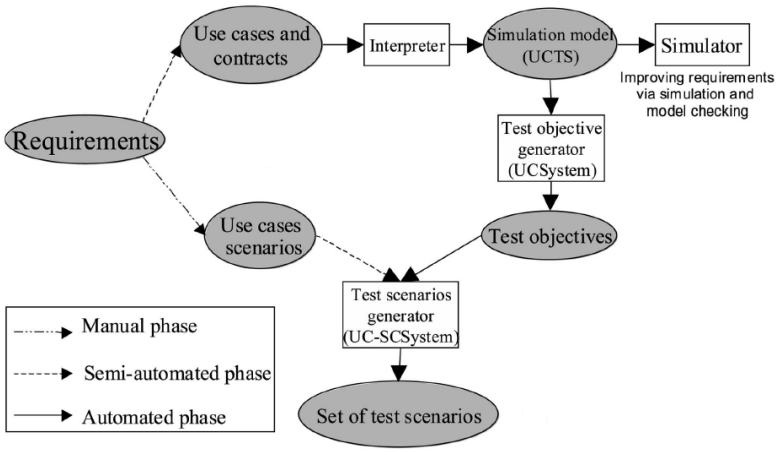

Figure 4.1: Flow of generating test scenarios [3]

application (deterministic postconditions). Parameters to use cases can either be system actors are main concepts of the use case, which are represented as normal instantiated classes during the test generation process.

Through exhaustive simulation by the prototype/interpreter-tool *UCSystem* a *use case transition system* (UCTS) is built, which serves as a model for all valid sequences of use cases. Therefore first 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 stored. The requirement engineer is here also given 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 amount of states in the transition system can be calculated with

$$maxsize_{UCTS} = 2^{n_{ip}} \tag{4.1}$$

where

$$n_{ip} = p \cdot max_{instances}^{max_{param}}. \tag{4.2}$$

n_{ip} describes the amount of instantiated predicates, p the amount of predicates, $max_{instances}$ the maximum amount of instances 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 criterion are mainly introduced to generate functional tests. The *robustness* criterion on the other hand explicitly exercises a use case in as many different ways to make its precondition false. Therefore valid test paths are generated, that 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 which rely on the rest of the model, they are written in *Object Constraint Language* (OCL), and there are contracts which 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 problem of the approach is that the simulation model has to be compact enough to avoid combinatorical 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 main system concepts and actors. Furthermore the above mentioned test objective generation criteria were identified through experimental comparisons and help to keep the amount of test objectives in a reasonably scope.

The generated test scenarios can either lead to a pass verdict, a fail verdict (in case a postcondition is violated) or a inconclusive verdict. The latter is invoced if a precondition evaluated to false ans the test scneario was non executed entirely. This could be because of uderspecification 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 4.3.

Table 4.3: 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 4.4.

Table 4.4: 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%

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 which 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 50% of the equivalent robustness code. Therefore the approach works good 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 deteiled preconditions of use case scenarios are not included in the test generation process.

One possible extensions of the approach was 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.

4.3.2 Application of Approach 1

In the first step the test objectives have to be derived. Therefore we define the use cases (can be thought of as main functions) and their contracts (Listing 4.1) as requirement-level 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.

Listing 4.1: Contracts attached to use cases

```
UC createMovie(m: movie)
post createdMovie(m)

UC createLinkedPerformer(p: performer, m: movie)
pre createdMovie(m)
post createdPerformer(p) and createdLink(p,m)

UC rateMovie(m: movie)
pre createdMovie(m)
post calculatedOverallRating(m)

UC ratePerformer(p: performer)
pre createdPerformer(p)
```

```
post forall(m: movie){ createdLink(p,m)@pre implies calculatedOverallRating(m) }

UC linkExistingMovie(m: movie, p: performer)
pre createdMovie(m) and createdPerformer(p)
post not createdLink(p,m)@pre implies (createdLink(p,m) and
    calculatedOverallRating(m))

UC linkExistingPerformer(m: movie, p: performer)
pre createdMovie(m) and createdPerformer(p)
post not createdLink(p,m)@pre implies (createdLink(p,m) and
    calculatedOverallRating(m))

UC unlinkMovie(m: movie, p: performer)
pre createdMovie(m) and createdPerformer(p) and createdLink(p,m)
post calculatedOverallRating(m) and not createdLink(p,m) and not exists(m2: movie){
    createdLink(p,m2) }@pre implies not createdPerformer(p)

UC unlinkPerformer(m: movie, p: performer)
pre createdMovie(m) and createdPerformer(p) and createdLink(p,m)
post calculatedOverallRating(m) and not createdLink(p,m) and not exists(m2: movie){
    createdLink(p,m2) }@pre implies not createdPerformer(p)

UC removeMovie(m: movie)
pre createdMovie(m)
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)

UC removePerformer(p: performer)
pre createdPerformer(p)
post not createdPerformer(p) and forall(m: movie){ not createdLink(p,m) and
    calculatedOverallRating(m) }
```

After that the UCSystem prototype/interpreter tool should build the UCTS (Figure 4.2) through exhaustive simulation. The pool of parameters was restricted to one movie and performer to avoid a combinatorical 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.

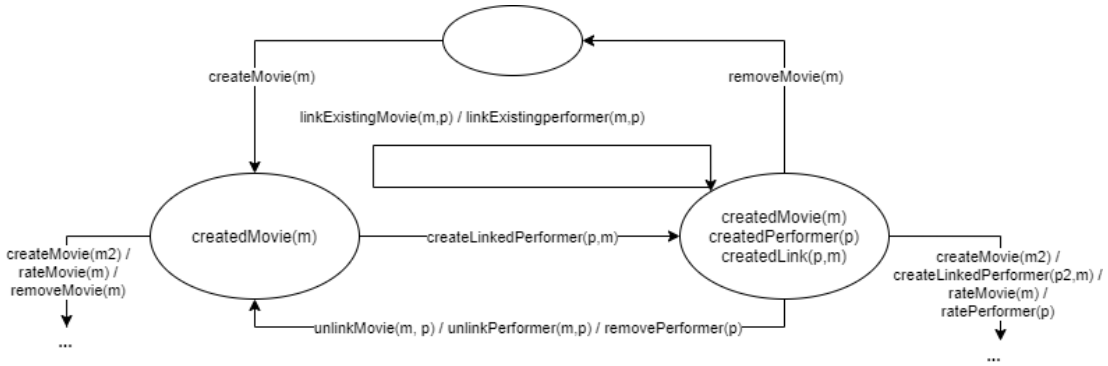

Figure 4.2: The Use Case Transition System

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 criterions. How many test objectives are derived depends on the internal implementation of UCSystem and cannot be predicted for this example. Let's assume that one test objective is the test path createMovie(m) -¿ createLinked-Performer(p,m) -¿ removeMovie(m). Then the use case scenarios from Figure 4.3 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.

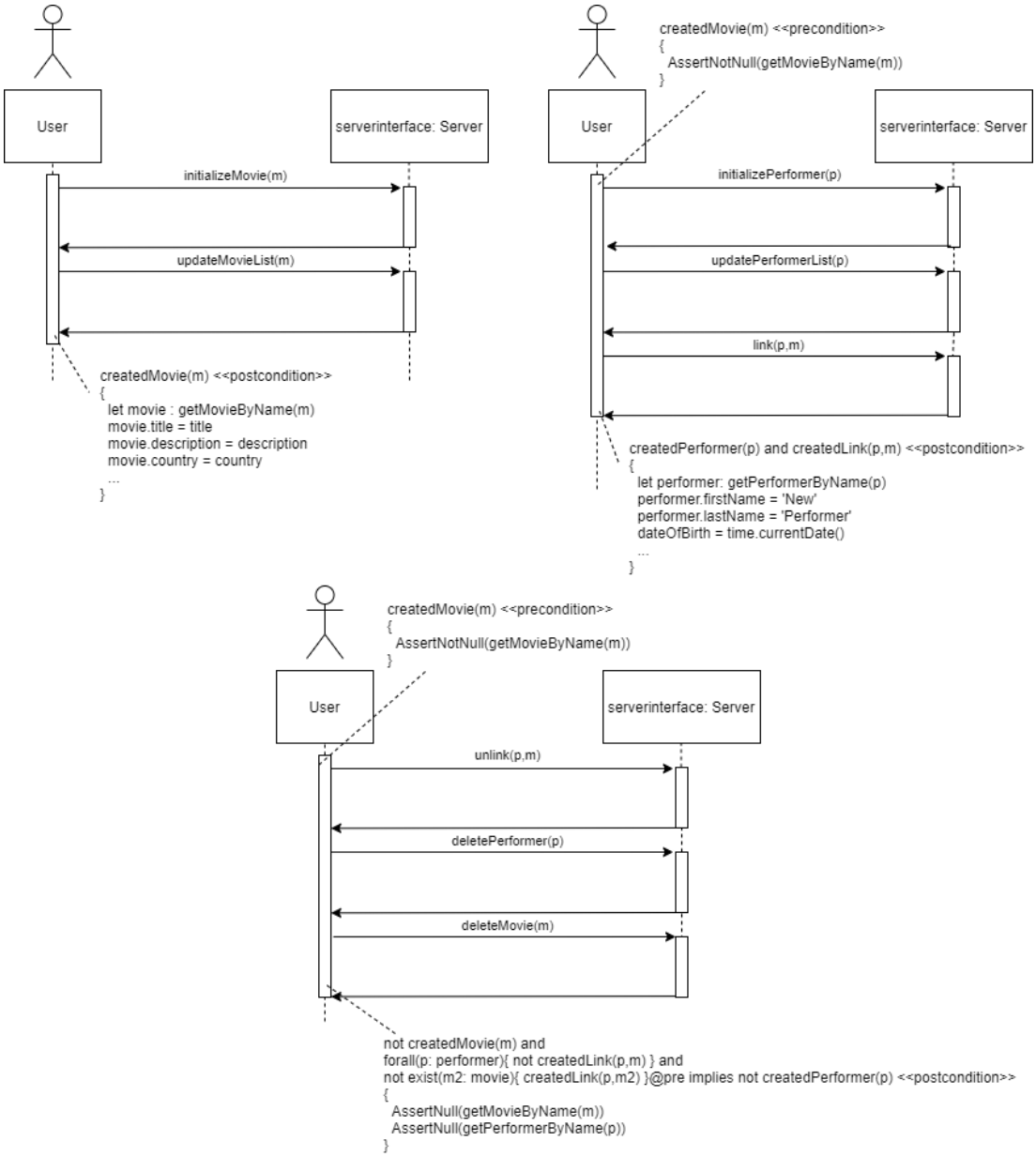

Figure 4.3: The Use Case Scenarios

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

4.4 Approach 2

4.4.1 Description of Approach 2

Based on the suggested improvements in the first paper, the second paper uses 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 early stages of development. Each node in the IOD represent either an interaction diagram (sequence diagrams) or interaction occurences 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 occurences) in the IOD. The CTS is built by a tool using the 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. If contract statements are the same for different operations, the identical statements are also split into two sub-states. Additional transitions are added for all conditional flows leading to alternative scenarios and their guard conditions are attached with them. Additional CTS flows help to further refine the requirements, potential unwanted behavior or underspecification can be found. 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).

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 but to IODs. It therefore serves as a platform to generate more in-depth test scenarios (as well for negative test cases). The paper does not provide additional steps on how the generate executable test scenarios from the retrieved test paths.

4.4.2 Application of Approach 2

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 4.4) with separate contracts defined in an OCL file (Listing 4.2). 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 contract transition system. The states are represented by the contracts themselves.

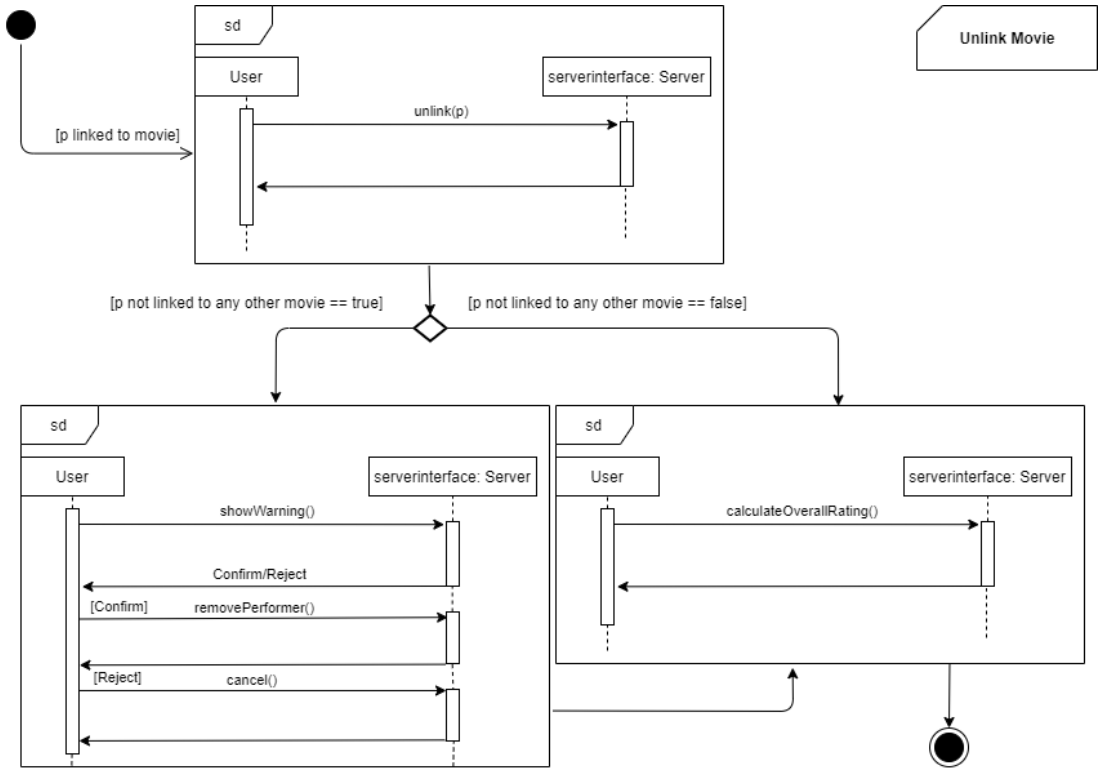

Figure 4.4: Interaction Overview Diagram

Listing 4.2: 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.rating = 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 4.5.

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_4	S_2	S_5	

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

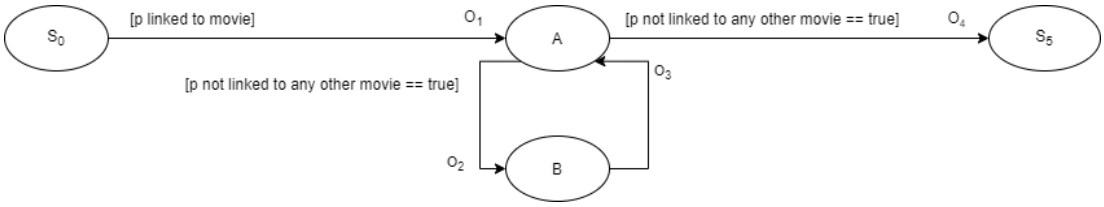

Figure 4.5: Contracts Transition System

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

4.5 Comparison

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 (Who 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?

Nr.	Approach [3]	Approach [4]
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 aquired 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 from. 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. UCSystem as a third party tool to build the UCTS and to derive test objectives using 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. UC-SCSystem to generate executable test scenarios.	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 use case transition system (UCTS). To derive test objectives the transition system is traversed according to one of the predefined coverage criteria All Edges (AE), *All Vertices* (AV), *All Instantiated Use Cases* (AIUC), *All Vertices and All Instantiated Use Cases* (AV-AIUC) or *All Precondition Terms criterion* (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 genereted by the 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 occurences 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, UCSystem 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 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 UCSystem. 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 updates 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.

Nevertheless the two approaches differ in their field of application. While [3] uses use cases to automatically generate system level tests, [4] restricts to generate test paths for a specific use case. It therefore does not need additional use case scenarios as an input like [3] to generate test scenarios as these are already natively given in the IOD.

[3] has the special characteristic that it demonstrates an end-to-end test scenario generation process until concrete test execution, which is missing in [4]. Nevertheless this could easily be implemented using similar tool support in [4] as well.

4.6 Conclusion

Both approaches deliver 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 through traversing a transition system.

In literature search a second paper was found that follows the process of automatically gener-

ating test paths through traversing a transition system, but this time uses the in [3] 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 where used, whereas the choice of relevant articles was based on previously specified relevance criteria. Eight resulting papers were found according to these criteria, [4] was finally chosen.

Approach [3] 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 most efficient and most extensive measured by statement coverage. Giving contract enriched use case scenarios as a second input helps to generate concrete executable test scenarios.

[3] switches from system level to use case level test generation. It uses contract enriched IODs as its 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 [3] can be used).

One weakness of both approaches remains the capability to generate a sufficient amount 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.

5 Conclusion

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

6 Bibliography

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

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

[3] Nebut, C., Fleurey, F., Le Traon, Y., Jézéquel, J.-M.: Automatic Test Generation: A Use Case Driven Approach. In: IEEE Transactions on Software Engineering (Volume: 32, Issue: 3, March 2006)

[4] Raza, N., Nadeem, A., Iqbal, M. Z. Z.: An Automated Approach to System Testing based on Scenarios and Operations Contracts. In: Seventh International Conference on Quality Software (QSIC 2007)

[5] Hausberger, F.: Research planning, research and mid-term presentation. https://github.com/fidsusj/SWE-Seminar

List of Figures

List of Tables