Heidelberg University
Institute of Computer Science
Software Engineering Group

Better Software Through Systematic Testing

Literature Synthesis

`https://github.com/fidsusj/SWE-Seminar`

Felix Hausberger, 3661293,
Applied Computer Science
eb260@stud.uni-heidelberg.de

1 Synthesis

1.1 Summary

Automatic Test Generation: A Use Case Driven Approach. The paper addresses the traceability problems between high-level use cases and concrete test case execution as well as the lack of integration of formal specification methods with well-established development life cycles. It shows a method to automatically generate executable test case scenarios by making use of a transition system and therefore shifting the effort of test generation to the specification activity. In the first step, UML use cases get enhanced with contracts (pre- and postconditions). The contracts are made executable by writing them in the form of requirement-level logical expressions. Through exhaustive simulation a transition system is built, which serves as a model for all valid sequences of use cases. Relevant test objectives get extracted from the transition system by applying predefined coverage criteria. Subsequently in the second step, test scenarios get generated by replacing each use case in a test objective with a use case scenario that is compatible in terms of static contract matching. The process results in executable test scenarios that get evaluated using the statement coverage metric.

An Automated Approach to System Testing based on Scenarios and Operations Contracts. 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. The IODs get enhanced with contracts written in the object constraint language (OCL) and the get transformed into a contracts transition system (CTS) which models all scenarios of the IOD. Through traversing the CTS test paths get derived. 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).

1.2 Synthesis matrix

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 [1]	Approach [2]
3a	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. The transition system (UCTS) itself as a simulation model to derive test objectives from. Test objectives describing the test paths. UC-System 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. 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. 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.
3b	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

3c To express the ordering constraints between use cases, each use case is attached by a contract. 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. 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 (UTCS). 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). 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. To build test scenarios a use case scenario can replace a use case at a certain stage of execution iff 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.

4a	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
4b	Test writers / Developers, Requirement Engineers	Test writers / Developers, Requirement Engineers
4c	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)
5a	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
5b	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
6a	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

6b	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.

2 Example

In this chapter both approaches should be demonstrated within the example of a movie management software.

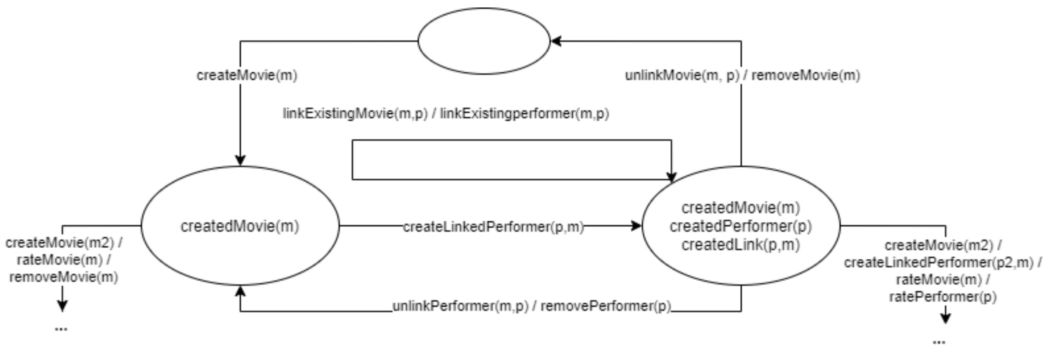
2.1 Automatic Test Generation: A Use Case Driven Approach

In the first step the test objectives have to be derived. Therefore we define the use cases and their contracts as requirement-level logical expressions. 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 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) }
```

After that the UCSystem tool should build the UCTS (2.1) through exhaustive simulation. The pool of parameters was restricted to one movie and performer to avoid a combinatorical explosion for this example. 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.


createMovie(m)
unlinkMovie(m, p) / removeMovie(m)
linkExistingMovie(m,p) / linkExistingperformer(m,p)
createdMovie(m)
createLinkedPerformer(p,m)
createdMovie(m)
createdPerformer(p)
createdLink(p,m)
createMovie(m2) /
rateMovie(m) /
removeMovie(m)
...
createMovie(m2) /
createLinkedPerformer(p2,m) /
rateMovie(m) /
ratePerformer(p)
...
unlinkPerformer(m,p) / removePerformer(p)

Depending on the selected coverage criterion, we receive different test objectives. 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) -> createLinkedPerformer(p,m) -> unlinkMovie(m).

2.2 An Automated Approach to System Testing based on Scenarios and Operations Contracts

References

[1] Clémentine Nebut, Franck Fleurey, Yves Le Traon, Jean-Marc Jézéquel, “Automatic test generation: A use case driven approach,” vol. 32, 2006. [Online]. Available: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1610607` (visited on 12/06/2020) (cit. on p. 3).

[2] Najla Raza, Aamer Nadeem, Muhammad Zohaib Z. Iqbal, Ed., *An Automated Approach to System Testing based on Scenarios and Operations Contracts*, IEEE, 2007. [Online]. Available: `https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4385504` (visited on 12/06/2020) (cit. on p. 3).