An Analysis of Use Case Based Testing Approaches Based on a Defect Taxonomy

Timea Illes[1], Barbara Paech[1]

[1] University of Heidelberg, Institute of Computer Science
Im Neuenheimer Feld 326
Germany-69120 Heidelberg
{illes, paech}@informatik.uni-heidelberg.de

Abstract: Use cases are a well-established means for requirements elicitation and specification. Recently, several approaches have argued to take use cases also directly as the basis for testing. In this paper we analyze use case based testing approaches on the basis of a defect taxonomy. For this purpose, we propose a taxonomy classifying typical defects which need to be uncovered during system testing. Then, we survey current approaches to derive test cases from use cases and discuss their ability to reveal these defects.

1 Introduction

Since their original introduction in [15], use cases (UC) have gained an increasing popularity. They are a well-established means for requirements elicitation and specification, modeling the behavior of a system from the user's point of view.

Recently, several approaches have been proposed which take UCs as input for test case development. The need to employ documented requirements as a basis for testing has already been recognized in the year 1979 [19]. A more recent survey insists on the necessity of using UCs as a basis for system testing [28]. UC based testing claims to offer a lot of advantages. One of these advantages is that UCs are widely used as inherent part of most object oriented analysis and design methodologies. Furthermore, the use of UCs as a basis for both, for software development as well as for testing, provides a uniform notation and a high reusability of requirements engineering artifacts. Additionally, the integration of testing activities into early development stages is alleviated. Finally, the development of test cases in parallel to UCs enables an early validation of the requirements.

But how well can these approaches support system testing? In order to answer this question, this paper examines which typical defects can be revealed during system testing and discusses the ability of current approaches to reveal the identified defect classes. The contribution of this paper is three-fold. First, we propose a defect classification for system level tests. Then, we evaluate current approaches for UC based testing with respect to their ability to reveal these defect classes. Finally, we add a testing perspective to requirements engineering (RE). The defect classes show how testers think about requirements and systems and what kind of information they need.

Related work. In [12] four approaches addressing the derivation of test cases from requirements are compared. Only two of them are based on UCs. Furthermore, the comparison is very superficially based on criteria such as the use of standards or the availability of a tool supporting the approach. In [2] an overview of the approaches to test case generation during RE is given. In contrast to this paper, the authors do not focus on UC based testing techniques and consequently they do not consider all approaches discussed in this paper. Additionally the comparison of the approaches is ad-hoc without a systematic definition of criteria.

Overview. The remainder of this paper is organized as follows. Section 2 starts with a brief introduction to the basic concepts of UC based testing. Section 3 introduces the defect taxonomy. Section 4 gives an overview of current approaches for UC based test case derivation and discusses how well they address the defect classes proposed in Section 3. Section 5 concludes the paper.

2 UC Based Testing – The Overall Approach

This section introduces some basic concepts. We explain the notions of UCs, of system testing and UC based testing. Additionally, we give an overview on UC based testing approaches considered in this paper.

2.1 Terminology

In the context of this paper we define UCs, based on the definition proposed in [22] as follows: *A UC is a sequence of steps executed cooperatively by the system (system steps) and outside actors (actor steps) in order to yield an observable result to the actor(s), including alternatives and exceptions*. Consequently, UC descriptions typically contain information on *tasks or goals (*Which tasks/goals of the actor(s) should be fulfilled by the UC?*), actors* (Who initiates/participates in the UC?), *preconditions* and *postconditions* (Which conditions have to be fulfilled before respectively after the UC execution?) as well as *actor steps* (actions to be performed by the actors, including input data) and *system steps* (actions to be performed by the system, including output data). Optionally, information on *rules* (describing complex functional or causal interrelations) as well as on *quality requirements* (e.g. usability or performance) can be added to the UC description.

According to the definition proposed in [13], *system testing* is concerned with the process of testing an integrated system in order to verify that it meets the specified requirements. For this purpose a finite set of test cases has to be developed, in order to execute the system under test (SUT) with different inputs. A *test case* contains a set of input values, execution preconditions, expected results and execution post conditions.

UC based testing is an approach to system testing, where test cases are defined and selected on the basis of the requirements specified in terms of UCs. Therein, UCs play different roles:

During testing, actual behaviour is compared with the expected behaviour in order to decide, whether a test was passed or not. The source to determine the expected behaviour of the SUT is called test oracle. Consequently, the UC specification serves as

test oracle in UC based testing, i.e. the UC specification is the source to define the expected output and post conditions as a result of the input and preconditions defined in a certain test case. If the actual behaviour corresponds to the expected behaviour, the SUT meets the specified UC. Since complete testing is impossible, a finite set of test cases has to be selected according to some coverage criteria indicating which parts of the SUT should be executed. Coverage criteria are determined according to a coverage item. In the case of UC based testing, UCs serve as *coverage items*. A weak coverage criterion is e.g. UC coverage, which requires at least one test case per UC. A stronger coverage criterion is e.g. path coverage which requires at least one test case per UC path.

2.2 Considered Approaches

Table 1 gives an overview on the approaches and the particular models into which UCs are transformed.

Table 1. Overview of the approaches and corresponding models

Approach (ID, Name)		Model Transformation
A	Path Analysis [1]	UC, no transformation
B	Testing with UCs [24]	State charts, Activity Diagrams
C	Extended UCs [4]	Tabular representation
D	Requirements by Contracts [21]	UC transition system (nodes: system states, transitions: instantiated UCs)
E	TOTEM [6]	Activity Diagrams, Sequence Charts, regular expressions
F	SCENT [25]	Annotated state charts, dependency charts
G	Simulation and Test Model [29]	Extended interaction overview charts, state charts
H	Purpose Driven Testing [3]	Goal Graphs (different abstraction level)
I	ASM based Testing [11]	Abstract state machines

For our analysis we selected approaches according to the following criteria:

(a) The approaches are based on UC descriptions or UC diagrams
(b) For each of the following approach classes we selected representative approaches.

Model exploration approaches exploit the information contained in UCs *as is*. Most approaches of this class are white papers. We selected the Path Analysis approach (A in Table 1) because it was the only approach of this class mentioning the GUI.

Model extension/transformation approaches extend the information contained in UCs by test related information. Additionally, an informal or structured UC model is transformed into a semi-formal, mostly graphical model. When appropriate, the resulting model is retransformed into a new model. On the basis of the resulting model, test cases are (semi-) automatically derived. For this purpose the models are traversed according to some coverage criteria, where a path usually corresponds to a test case. The approaches B-H in Table 1 belong to this class. We selected the approaches so

that all target models (e.g. state charts or a proprietary model) are represented. Additionally we included the approach B in Table 1 as it addresses inter-software defects.

Model formalization approaches take an informal or structured UC model as input and transform this into a formal model. On the basis of this model, test cases can be automatically generated according to specific coverage criteria defined for that model. As a representative of this class we selected the ASM Based Testing approach (I in Table 1).

3 Defect Taxonomy

We now identify typical defect classes which need to be uncovered during system testing. We based our defect classification on taxonomies proposed in [5, 16]. In contrast to these defect taxonomies, which address defect classes at different phases of the development life cycle, e.g. defects in the requirements specification document, we restricted our taxonomy to defects which can be detected during *system testing*. Additionally, we refined the resulting taxonomy by analyzing further defect classifications like those proposed in [17, 18, 27] with respect to their applicability for system testing. In contrast to our taxonomy, these classifications have a particular focus on e.g. defects in e-commerce applications [27] or taxonomies for security issues [17] and [18]. Finally, we validated our taxonomy by investigating, to what extend defects captured in bug reports for open source software can be classified according to our taxonomy. For this purpose we investigated several bug reports stored in the bug tracking system of the mozilla.org [7] database. The defects recorded in this database refer to software such as the web browser Firefox [10], the Email Client Thunderbird [26] and other mozilla.org projects [20]. Due to the comprehensiveness of the database, we only considered "blocker", "critical" and "major" defects. Additionally we investigated defect lists of two open source CRM (Customer Relationship Management) projects [9, 23]. The result is a list of defect classes for system testing. Each defect class can be refined by subclasses. The defect classes are not orthogonal, i.e. a defect can be categorized into more than one defect class. Additionally, a defect can also be associated to a combination of defect classes. In the following, we present a short definition and corresponding examples of typical subclasses for each defect class.

Completeness defects subsume all defects related to an incomplete implementation of the specified functionality. Typical defects in this class are *missing functionality defects* (the implementation of a specified or desired requirement is missing) and *undesired functionality* (additional, undesired functionality has been implemented). There are two typical defects which can occur in the presence of additional, undesired functionality: *prevention defects* (if additional functionality prevents the execution of the desired functionality) and *overlapping defects* (if additional functionality and desired functionality overlap).

Input/Output defects subsume all defects related to wrong input respectively to wrong output data of the SUT. Typical input/output defects include *boundary defects* (e.g. date < 21.02.2006 instead of date < 13.02.2006), defects concerning *wrong size, shape or format* of the data or *combination defects* (i.e. defects which occur, when certain input values respectively output values are combined).

Calculation defects subsume all defects resulting from wrong formula or algorithms in the SUT (e.g. defect in the search algorithm: The system looks for product descriptions, containing *all* of the key words entered by the customer, instead of finding also products containing *at least one* of the entered keywords).

Data handling defects subsume all defects related to the lifecycle and the order of operations performed on data. Typical data handling defects include *duplicated data* (e.g. system fails when creating duplicated data) or *data flow defects* (defects related to the sequence of accessing a data object (e.g. data update before the data has been created).

Control flow and sequencing defects subsume all defects related to the control flow or the order and extent to *which* processing is done, as distinct from *what* is done [5]. Typical control flow defects concern *wrong sequencing* of the actions performed or *iteration and loop defects,* which subsume all defects related to the control flow of iterations and loops.

Concurrency defects subsume all defects related to the concurrent execution of parts or of multiple instances of the SUT. Typical defects contained in this class include priority defects and race condition defects. *Priority defects* are related to the assignment of a wrong priority (too high, too low, priority selected not allowed), e.g. a phone call on a mobile phone does not pre-empt the execution of an arbitrary function when a phone call has been received). *Race condition defects* are related to the competition of processes for a limited resource, e.g. for time, or for shared data.

GUI defects subsume all defects related to the user interface, which are not usability defects. Typical defects of this class are *display defects* (defects related to the display and highlighting of the information on the screen, e.g. failure to clear or update part of the screen or failure to clear highlighting) and *navigation defects* (e.g. missing or disabled menu entries).

NFR (non-functional requirement) defects subsume all defects related to the quality of the SUT. According to [14], defects concerning *functionality*, *reliability, usability, efficiency, maintainability and portability* belong to this category.

Inter-Software defects subsume all defects concerning the interface of the SUT to other software systems. Typical defects of this class are *input/output defects* (if there is a syntactic or semantic misunderstanding between the interacting software systems), *concurrency defects* (e.g. if the SUT and a COTS component compete for the same data) or *completeness defects* (e.g. if functionality of the third party software is missing).

Hardware defects subsume all defects concerning the interface of the SUT to the hardware. Typical defects of this class are *input/output defects* (e.g. incorrect interpretation of returned status data).

4 Evaluation of the Approaches

UCs are intuitive, informal and thus easily readable for different stakeholders. Consequently, UCs are well suited in the context of requirements elicitation and specification. However, when UCs are used as a basis for test case derivation the perspective changes. In this case, the stakeholders of the UC specification are testers, who aim to

find defects in the SUT. The aim of this paper is not the evaluation of the UC concepts itself, but the efficiency of UC based testing techniques.

Based on the defect taxonomy introduced in Section 3, we now discuss the defect classes with respect to their ability to be revealed by UC derived test cases. Additionally, for each defect class typical solutions are presented. Table 3 summarizes the result of our analysis. A „+“ indicates that the corresponding defect class is well addressed by an approach, a “(+)” indicates that the corresponding defect class is partially considered (e.g. parts of the possible defects in the defect class are addressed). A “-“ indicates that the approach does not consider the corresponding defect class at all.

In order to assure comparability of the approaches, we assume a correct UC specification and evaluate the efficiency of the techniques with respect to a given correct specification. All techniques assume a correct requirements specification because the test case set derived is as good as the UCs themselves. Some approaches give guidance for the specification and validation of use cases. But this aspect is not part of our evaluation. Furthermore, to assure an efficient evaluation, we focus on defects which can be associated with a single defect class and do not consider defects which result by all possible combinations of different defect classes.

4.2 Completeness Defects

In general missing UC implementation is revealed easily on the basis of a UC specification. Most approaches will uncover a missing UC implementation, since they iterate over all UCs and perform some analysis *per* UC, e.g. determine all paths within a UC or develop a new model e.g. a state chart representation *per* UC. Consequently, there is at least one test case per UC which would detect the missing implementation of a UC. Whether missing parts of an UC can be uncovered, depends on which coverage criteria the corresponding approach defines, e.g. path coverage will easily uncover a missing case within a UC. Coverage criteria will be discussed along with control flow and sequencing defects. As the approaches [21] and [3] focus on the *interaction* between UCs, they are not well suited to uncover missing parts *within* a UC.

4.3 Input / Output Defects, Calculation and Data Handling Defects

The detection of input/output defects, calculation defects as well as data handling defects depends on the accuracy with which the respective details have been documented. Due to the fact that UCs are typically phrased in natural language, they are imprecise. Therefore, test cases derived from UCs will hardly reveal *input/output defects*, *calculation defects* as well as *data handling defects*.

Input/Output Defects. In [4] the concept of extended UCs is introduced. Extended UCs express the relationship between system state (precondition of a UC), a combination of inputs and the expected results in terms of a decision table. For each combination of inputs and system state which results in distinct classes of SUT behaviour a new relation in terms of a new row in the decision table is defined. Then, test cases are derived using combinatorial strategies. Following this approach, input/output de-

fects can easily be uncovered. A light weight approach is the annotation of UCs or the models derived on the basis of UCs with test related data including input values or possible ranges for the input or output data. This is the case in [25] and [3].

Calculation Defects. Calculation defects are not addressed by any particular approach especially. However, the approach proposed in [4] is suited best for this purpose. The tabular representation, relating a combination of inputs and system states to outputs can easily be adapted to the creation of test cases, which test e.g. a formula specified within a UC with different input combinations.

Data Handling Defects. In [6] the life cycle of a „business object" and related defects are addressed by representing the life-cycle of these objects in terms of activity diagrams, which relate UCs to each other. The UCs are grouped into swimlanes, where each swimlane represents the life cycle of a business object from its creation until its deletion. UCs grouped into the same swimlane manipulate (read, write) the corresponding object. Valid sequences of UCs are generated by traversing the activity diagram. A path in the activity diagram represents a test case, and thus a possible life-cycle of a business object. In [21] pre and post conditions of a UC are expressed in terms of contracts on the inputs respectively on outputs of a UC e.g. an item has to be created so that the UC delete item can be executed. Thus, sequences in the life cycle of business objects can be created by concatenating UCs so that the post condition of one UC represents the precondition of the next UC. However, both approaches consider only valid paths. Negative test cases, e.g. which test unwanted behaviour are not created.

4.4 Control Flow and Sequencing Defects

Sequences of interaction between user and system as well as alternatives and exceptions within a UC can easily be expressed. Hence, control flow as well as sequencing defects in the implementation of that particular UC can easily be detected. But since UCs comprise self-contained coherent units of functionality, they are not suited to express the interplay between distinct UCs. Consequently, test cases which verify the correct implementation of the interaction *between* UCs are hard to be derived from UC specifications.

Some approaches [1] and [8] require structural coverage of UCs by test cases, e.g. path coverage. Thus, each path in a UC is executed by at least one test case. Consequently, these approaches will likely reveal control flow defects *within* the implementation of a UC. In [1] all paths of a UC are required to be uncovered by a test case. In [8] test cases are derived which exercise all combinations of executing and non-executing an <<extends>> relationship. Most approaches transform the UC model into another, more formal model e.g. a state chart or a sequence diagram representation and require coverage of the new model. This is the case in [24, 6, 25, 29 and 11]. Usually the models are then traversed according to coverage criteria of the new model. As the transformation into a new model is not automatic, there is a risk not to consider all information defined in a UC, and thus, not to detect all defects which would be detected based on the original UC specification.

Control flow defects in the implementation of the interaction *between* UCs are especially addressed in [3, 6, 21, 24, 25 and 29]. In [3] the interaction between UCs re-

alizing a user goal is addressed. In [6] valid sequences of UCs are expressed in terms of activity diagrams. Test cases are derived by traversing all valid sequences. In [21] contracts on the execution of a UC are defined by expressing pre and post conditions of a UC. On the basis of these contracts, a transition model of valid UC sequences can be defined by concatenating post conditions of a UC with the precondition of another UC. Test cases are generated from the transition model according to given coverage criteria. In [24] state models derived from single UCs are merged by "composition". Test cases are then derived by covering all valid state combinations in the "composed" state model. In [25] the interaction between UCs is expressed in a new diagram type, the so called "dependency chart". Dependency charts can express dependencies between scenarios, e.g. sequential dependencies, alternatives or iterations. The authors use the term "scenario" equivalent to the term "use case". Test cases are derived from dependency charts mainly by trying to break the constraints defined. The authors give advice on how to break these constraints. In [29] sequential dependencies between UCs are identified and represented in terms of an UML interaction overview diagram. This diagram is then transformed into a state chart model which is traversed in order to derive test cases for each path in the state chart. No approach, except the one introduced in [25], considers invalid paths and the systematic derivation of test cases for trying to execute invalid paths.

Table 2 summarizes the evaluation of the approaches according to their efficiency to detect control flow defects *within* the implementation of a UC and respectively in the implementation of the *interaction between* UCs. Approaches which consider both defect subclasses are highlighted in light grey.

Table 2. UC based testing approaches and control flow defects

	Path analysis [1]	Purpose Driven Testing [3]	Extended UCs [4]	TOTEM [6]	Structural Testing with UCs [8]	ASM Based Testing [11]	Requirements by Contracts [21]	Testing with UCs [24]	SCENT [25]	Simulation and Test Models [29]
Control flow defects within the implementation of a UC	+	-	-	+	+	+	-	+	+	+
Control flow defects in the implementation of the interaction between UCs	-	+	-	+	-	-	+	+	+	+

4.5 Concurrency Defects

Expressing constraints on the parallel execution is not supported by UCs. Thus, UC derived test cases will hardly uncover concurrency defects.

Concurrency defects are addressed in [29] by defining dependencies between UCs and documenting these dependencies in terms of UML interaction overview diagrams. The dependencies concerning constraints on the parallel execution cover: *parallel execution* (when two or more UCs can be executed in parallel), *preemption/suspension* (when one UC pre-empts the execution of another UC having a higher priority), *exclusion* (when a UC can not be executed during the execution of

another UC) and *multi-instantiation* (when multiple instances of a UC may be executed at once). Exclusion and suspension are not part of UML 2.0 interaction overview diagrams. Thus, two additional stereotypes have been added to denote the corresponding relationships. The approach also contains a methodology to transform these diagrams into state machines which are traversed and covered in order to obtain test cases. Concurrency is also addressed in [25]. Dependency charts can express constraints on the parallel execution of scenarios. Thus, enforced, prohibited as well as an accidental parallelism can be expressed in terms of relationships between scenarios. Furthermore, constraints on the starting time (scenarios have to start/end at the same time or scenarios have to start one after the other with a given time interval between them) as well as data/resource dependencies can be included. Test cases for each identified dependency have to be developed. The authors propose to focus on "unwanted" behaviour by defining test cases which try to break dependency constraints. In [24] the necessity of modelling parallelism is stated, but how this should be expressed and how corresponding test cases should be derived is not explained. In [6] it is possible to express that the executions of two UCs are independent of each other. But, there is no advice on how to derive test cases which address parallelism.

4.6 GUI & NFR Defects

UCs specify the functional requirements for a system, i.e. they indicate "*what*" should be realized by the system, in contrast to non-functional requirements, which describe *"how well"* a requirement should be realized. The latter can not be expressed well in terms of UCs. Furthermore, UCs abstract from a specific user interface, they specify e.g. that an actor initiates a particular function, but they do not define whether this action occurs by clicking on a hyperlink of a web interface or by selecting a menu item from a windows-based system. Consequently, UCs are not well suited to derive test cases for this class.

GUI defects are considered solely in [1] and [25]. In these approaches, GUI-related information can be annotated to test cases [1] or to intermediate models derived from UCs [25]. The authors do not illustrate how this information can be systematically used to develop (further) test cases.

Non-functional defects, especially performance defects are addressed in [25] only. According to this, the semi-formal models developed on the basis of scenarios are annotated with non-functional requirements. When test cases are derived by covering the state chart, these requirements have to be considered.

4.7 Inter Software Defects

UCs abstract from the internal realization of the functionality, more precisely they describe the functionality without specifying, whether it will be realized by the SUT or by a third party component. Accordingly, the defects detected by UC derived test cases are mostly independent of the realizing (sub)system. A missing case within the implementation of a UC will e.g. be detected by a test case derived from this UC, in-

dependent of the realizing (sub)system. There are, however particular cases, which have to be considered.

The first case concerns *undesired functionality* of a third party system. In the case of COTS-software, which is intended to be used in different contexts, the functionality provided is often much more comprehensive than the functionality needed in the context of the SUT. Consequently, prevention defects (e.g. settings in a web browser prevent the execution of parts of a web based application written in JavaScript) as well as overlapping defects (e.g. when the "back" functionality in a web browser and the "back" functionality in the web application interleave) are very likely to occur. The second particular case concerns *known defects* in third party software. These defects represent a special kind of control flow defects, namely exception handling defects, where the SUT has to deal with exceptions of third party software.

As the architectural decisions, as well as decisions concerning which components will be developed and which will be bought, occur at a later development stage as the development of the UCs, the information on additional functionality is not contained in UCs. Therefore, UC derived test cases will hardly uncover the defects mentioned before.

Inter software defects are addressed in [24] and in [25] in the requirements specification phase, where guidance is given on how to identify the interface of the SUT. According to this, all hardware interfaces as well as software interfaces to the SUT are identified. These interfaces are considered (and covered) during the system test. In [29] concurrency defects mainly in the context of distributed components of a software system are addressed. Nevertheless, none of these approaches deals with overlapping functionality or with known defects in the third party software.

Table 3. UC based testing approaches and addressed defect classes

	Path analysis [1]	Purpose Driven Testing [3]	Extended UCs [4]	TOTEM [6]	Structural Testing with UCs [8]	ASM Based Testing [11]	Requirements by Contracts [21]	Testing with UCs [24]	SCENT [25]	Simulation and Test Models [29]
Completeness	+	(+)	+	+	+	+	(+)	+	+	+
Input / Output	-	(+)	+	-	-	-	-	-	(+)	-
Calculation	-	-	(+)	-	-	-	-	-	-	-
Data Handling	-	-	-	(+)	-	-	(+)	-	-	-
Control flow/ Sequencing	(+)	(+)	-	+	(+)	(+)	(+)	+	+	+
Concurrency	-	-	-	-	-	-	-	-	+	+
GUI	(+)	-	-	-	-	-	-	-	(+)	-
NFR	-	-	-	-	-	-	-	-	(+)	-
Inter Software	-	-	-	-	-	-	-	(+)	(+)	(+)
Hardware	-	-	-	-	-	-	-	(+)	(+)	-

4.8 Hardware Defects

UCs not only abstract from the realization, but also from the underlying hardware and external devices. Indeed, most defects at the interface of the SUT occur in the hardware. But as stated in [16], a *software* defect will also occur, if the software system does not recognise and treat a defect in the hardware. Hence, test cases have to be defined, which address defects in the software concerning the exception handling of hardware defects. Since UCs do not contain information on hardware, UC derived test cases are not well suited to reveal this type of defects.

Similar to software defects, hardware defects are addressed in [24] in the requirements specification phase. Hardware interfaces to the SUT are identified and documented. These interfaces can be considered during system testing. In [25] dependency charts, an annotation can be associated to a causal dependency concerning constraints on hardware, e.g. a printer has to be connected before a particular scenario can be executed (e.g. printing a document).

5 Conclusion and Future Work

In this paper we identified defect classes and discussed their ability to be uncovered by UC based testing approaches. Control flow and completeness defects are addressed by almost all approaches. No approach proposes a methodology to enrich UCs with GUI and NFR related information and to systematically derive test cases for testing the GUI and non-functional requirements. SCENT [25] is the most comprehensive approach, addressing more defect classes than all other approaches which have been analysed. It is a lightweight approach for UC based testing which addresses most of the defect classes by annotating UC derived models with test related information. In order to define a middleweight and more thorough approach for UC based testing, some issues concerning the integration of the GUI and of NFRs must be considered. Our future work will address the definition of an integrated model for RE and test development which allows the detection of NFR and GUI defects. Furthermore, we aim at designing a thorough evaluation of the approaches according to a strong benchmark.

References

1. Ahlowalia, N.: Testing from Use Cases Using Path Analysis Technique, International Conference On Software Testing Analysis & Review, (2002)
2. Allmann, C., Denger, C., Olsson, T.: Analysis of Requirements-based Test Case Creation Techniques, IESE-Report No. 046.05/E, (2005), http://www.iese.fraunhofer.de/pdf_files/iese-046_05.pdf, last visited July 2006
3. Alspaugh, T.A., Richardson, D.J., and Standish, T.A.: Scenarios, State Machines and Purpose Driven Testing, 4th International Workshop on Scenarios and State Machines: Models, Algorithms and Tools (SCESM'05), St. Louis, USA, (2005)
3. Binder, R.: Testing Object-Oriented systems, Addison-Wesley, (2000)

4. Beizer, B.: Bug Taxonomy and Statistics, Appendix, Software Testing Techniques, Second Edition, Van Nostrand Reinhold, New York, (1990)
5. Briand, L., and Labiche, Y.: A UML-based Approach to System Testing, Technical Report, Carleton University, (2002)
6. Bugzilla, https://bugzilla.mozilla.org/, last visited July 2006.
7. Carniello, A., Jino, M., and Lordello, M.: Structural Testing with Use Cases, WER04 - Workshop em Engenharia de Requisitos, Tandil, Argentina, (2004)
8. Compiere, http://www.compiere.org/, last visited July 2006
9. Firefox, http://www.firefox.com/, last visited July 2006
10. Grieskamp, W., Lepper, M., Schulte, W., Tillmann. N.: Testable Use Cases in the Abstract State Machine Language, Second Asia-Pacific Conference on Quality Software (APAQS'01), (2001)
11. Gutierrez, J.J., Escalona, M.J., Mejías, M., Torres, J., Álvarez, J.A.: Comparative Analysis of Methodological Proposes to Systematic Generation of System Test Cases from System Requirements, Proceedings of the 3rd International Workshop on System Testing and Validation, (SV'2004), ISBN: 3-8167-6677, Paris, France, (2004), pp. 151-160
12. International Software Testing Qualifications Board, ISTQB Standard Glossary of Terms used in Software Testing V1.1, (2005)
13. International Standard ISO/IEC 9126, Information technology - Software Product Evaluation - Quality Characteristics and Guidelines for Their Use, International Organization for Standardization, International Electrotechnical Commission, Geneva, (1991)
14. Jacobson, I., Christerson, M., Jonsson, P., and Oevergaard, G.: Object-Oriented Software Engineering: A Use Case Driven Approach, Addison Wesley, (1992)
15. Kaner, C., Falk, J., and Nguyen, H. Q.: Testing Computer Software, 2nd Ed., Wiley, New York, (1999)
16. Krsul, I.: Software Vulnerability Analysis, Department of Computer Sciences, Purdue University, Ph.D. Thesis, COAST TR 98-09; (1998)
17. Lough M.L.: A Taxonomy of Computer Attacks with Applications to Wireless, PhD Thesis, Virginia Polytechnic Institute, (2001)
18. Meyers, G.J., The Art of Software Testing, John Wiley & Sons, New York, (1979)
19. Mozilla.org, http://www.mozilla.org/, last visited July 2006
20. Nebut, C., Fleurey, F., Le Traon, Y., and Jézéquel, J.-M.: Requirements by contracts allow automated system testing, Proc. of the 14th. IEEE International Symposium on Software Reliability Engineering (ISSRE'03), (2003)
21. Object Management Group. UML Superstructure Specification, v.2.0, (2005)
22. opentaps, http://www.opentaps.org/, last visited July 2006
23. Rupp, C., and Queins, S.: Vom Use-Case zum Test-Case, OBJEKTspektrum, vol. 4, (2003)
24. Ryser, J., and Glinz, M.: SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test, Technical Report, University of Zürich, (2000/03)
25. Thunderbird, http://www.mozilla.com/thunderbird/, last visited July 2006
26. Vijayaraghavan, G.: A Taxonomy of E-Commerce Risks and Failures. (Master's Thesis) Department of Computer Sciences, Florida Institute of Technology, Melbourne, FL, May 2002
27. Weidenhaupt, K., Pohl, K., Jarke, M., and Haumer, P.: Scenario Usage in System Development: A Report on Current Practice. IEEE Software, (1998)
28. Whittle, J., Chakraborty, J., and Krueger, I.: Generating Simulation and Test Models from Scenarios, 3rd World Congress for Software Quality, (2005)