A Scenario-Based Approach to Validating and Testing Software Systems Using Statecharts

Johannes Ryser **Martin Glinz**
Department of Computer Science
University of Zurich
Winterthurerstrasse 190
CH-8057 Zurich, Switzerland
+41-(0)1-63 54572 / +41-(0)1-63 54570
Fax: +41-(0)1-63 56809
ryser@ifi.unizh.ch / glinz@ifi.unizh.ch

ABSTRACT

Scenarios (Use cases) are used to describe the functionality and behavior of a (software) system in a user-centered perspective. As scenarios form a kind of abstract level test cases for the system under development, the idea to use them to derive test cases for system test is quite intriguing. Yet in practice scenarios from the analysis phase are seldom used to create concrete system test cases. In this paper we present a procedure to create scenarios in the analysis phase and use those scenarios in system test to systematically determine test cases. This is done by formalization of scenarios into statecharts, annotation of statecharts with helpful information for test case creation/generation and by path traversal in the statecharts to determine concrete test cases.

KEYWORDS

Scenario, use case, testing, scenario-based testing, statechart annotation

Presented at the 12th International Conference on Software and Systems Engineering and their Applications ICSSEA'99. Proceedings: CNAM, Paris, France.

1 INTRODUCTION

Validation and verification are generally recognized as two vital activities in developing a (software) system. Testing plays an important role in validating and verifying systems. Yet nowadays testing is often done in an ad hoc manner, and test cases are quite often developed in an unstructured, non-systematic way.

A proposed (and valuable) approach to solve the problem lies in automating testing and – often a prerequisite to automation – in specialized test languages and formal specifications. But up to date, only very limited tool support exists, and formal specifications/special test languages are expensive to apply. It is not possible to automate the whole testing process and achieve acceptable test coverage in given time for projects relying on natural language specifications [11, 15].

In this paper we propose the use of scenarios, not solely for requirements elicitation and specification, but specifically for system testing. We do so by capturing natural language scenarios, converting them into formal scenarios using statecharts, and deriving test cases from statecharts in a systematic manner. The presented method is easy to apply, integrates nicely with existing software development methods and does not impose an inappropriate overhead. Systematic test case development is supported and testing is taken up early in the development process. Furthermore, we utilize synergies between the phases of system analysis & specification and system test by reuse of scenarios.

The rest of the paper is organized as follows: Section 2 serves as an introductory chapter to present the main concepts of scenarios and define some of the terms used in this article. Our approach – the SCENT-Method – is shortly delineated. In section 3 we present the basic concepts and principles of the SCENT-Method, and describe the procedures of scenario creation, formalization and test case generation in more detail. Finally in section 4 we present some conclusions.

2 OVERVIEW OF THE SCENT-METHOD

In this section we shortly introduce some key concepts of scenarios and then present an overview of the SCENT-Approach.

2.1 Scenarios

The notion of scenarios is central to our approach: Scenarios are any form of description or capture of user-system interaction sequences.

We define the terms scenario, use case and actor as follows:

Scenario -	An ordered set of interactions between partners, usually between a system and a set of actors external to the system. May comprise a concrete sequence of interaction steps (instance scenario) or a set of possible interaction steps (type scenario).
Use case [8] -	A sequence of interactions between an actor (or actors) and a system triggered by a specific actor, which produces a result for an actor. A type scenario in our terminology.
Actor -	A role played by a user or an external system interacting with the system to be specified

2.2 The SCENT-Approach

Motivated by the need for testing methods that are apt for practice and that are integrated with existing development methods, we propose a scenario-based approach to support systematic test case development. We aim at improving and economizing test case development:

- by (re)using and utilizing artifacts from earlier phases of the development process, specifically of the analysis phase, in testing again, thus taking profit of synergies between the closely related phases of system analysis and test,
- by integrating the development of test cases in early phases of the development process; that is, by interweaving testing activities with the activities of the early analysis and design phases of the software engineering process,
- and by defining a method how to develop test cases systematically.

We call our approach the SCENT-Method – A Method for SCENario-Based Validation and Test of Software.

The key ideas in our approach are:

1. Use scenarios not only to elicit and document requirements, to describe the functionality and specify the behavior of a system, but also to validate the system under development while it is being developed,
2. Uncover ambiguities, contradictions, omissions, impreciseness and vagueness in natural language descriptions (as scenarios in SCENT are at first) by formalizing narrative scenarios with statecharts [5],
3. Annotate the statecharts – where needed and helpful – with pre- and post-conditions, data ranges and data values, and performance requirements, to supply all the information needed for testing and to make the

statecharts suitable for the derivation of actual, concrete test cases,

4. Systematically derive test cases for system test by traversing paths in the statecharts, choosing a testing strategy as appropriate and documenting the test cases.

These key concepts need to be supported by and integrated with the development method used to develop the application or the system, respectively. Most object-oriented methods support use cases and statecharts or a comparable state-transition formalism. Thus, the integration of the proposed method in any one of those methodologies is quite simple and straightforward.

3 BASIC PRINCIPLES OF THE SCENT-METHOD

The idea of using scenarios/use cases in testing is not new: Jacobson writes in his book and articles that use cases are well suited to be used as test cases for integration testing, but does not define a procedure [8, 9]. Others have taken up, formalized and extended the notion of using scenarios to test a system (e.g. [6] using regular grammars and deterministic finite-state machines, or [2] defining scenario lifecycle diagrams). Yet despite the ubiquity of scenario approaches, a practical method supporting testers in developing test cases from scenarios has not emerged yet.
The method presented here comprises three main parts: Scenario creation, scenario formalization and test case derivation. All three are described in more detail in the following sections.

3.1 Scenario Creation

Most scenario processes used today are lacking a step procedure for the creation and use of scenarios. For this reason we define a procedure to elicit requirements and document them in scenarios (see Table 1). We use a scenario description template to document and format narrative scenarios. Thus we enforce adherence to a common layout and structure.

Table 1: The scenario creation procedure

#	Step Description
1	Find all actors (roles played by persons/external systems) interacting with the system
2	Find all (relevant system external) events
3	Determine inputs, results and output of the system
4	Determine system boundaries
5	Create coarse overview scenarios (instance or type scenarios on business process or task level)
6	Prioritize scenarios according to importance, assure that the scenarios cover system functionality
7	Create a step-by-step description of events and actions for each scenario (task level)
8	Create an overview diagram and a dependency chart (see section 3.3)
9	Have users review and comment on the scenarios and diagrams
10	Extend scenarios by refining the scenario description, break down tasks to single working steps
11	Model alternative flows of actions, specify exceptions and how to react to exceptions
12	Factor out abstract scenarios (sequences of interactions appearing in more than one scenario)
13	Include non-functional (performance) requirements and qualities in scenarios
14	Revise the overview diagram and dependency chart
15	Have users check and validate the scenarios (Formal reviews)

Actors, in- and outputs and events (as determined in steps 1-3) are uniquely named. A glossary of terms including a description of all actors, in- and outputs and all events is created.
The coarse scenarios created in step 5 are short natural language descriptions of the interaction and do not feature a step-by-step description yet. In step 6, instance scenarios are transformed into type scenarios and scenarios are prioritized, thus allowing for release planning.
Validation activities are interspersed throughout the development process (see step 9 and 15).
A full description of the scenario creation method can be found in [12].

The scenario creation procedure is not a linear process as it might appear from Table 1. The steps are highly intertwined and activities are performed in parallel and iteratively.

During the process of scenario creation and refinement the developer notes relevant information for testing; these remarks may be abstract test cases, reminders what not to forget and what specifically to test for, and so on. These notes are used during testing to enhance the test cases that are derived from statecharts by path traversal. Non-functional requirements and qualities are documented in natural language or with other appropriate means (formulas, timing constraints, pictures, graphics, screenshots, sketches, ...).

What about the cost of the approach? Scenario definition has to be done in any approach employing scenarios. No additional expense is imposed by the step procedure defined in the SCENT-Method. The level of detail in scenarios may be limited or extended to any desired depth, but restricting the details bounds the use of the scenarios for system design and for test case generation. On the other hand, there are some advantages to scenario specifications: scenario creation helps to get the user involved in the specification process and increases the domain and system understanding of the developers, thus enabling more accurate system modeling as well as better design and implementation. Scenarios facilitate better understanding and communication between developers and users/customers.

3.2 Scenario Formalization and Annotation

Validation of narrative scenarios by users is accompanied by verification steps by the developer. Verification is supported in the SCENT-Method by formalization: By converting natural language scenarios into statecharts, many omissions, ambiguities and inconsistencies can be found.

3.2.1 The Formalization Step

The formalization step is the transformation of structured natural language scenarios into a statechart representation. Contrary to other approaches ([3], [14]) we do not restrict and formalize natural language to capture requirements. Instead we rely on the developer to synthesize statecharts, supporting him/her with heuristics.
The heuristics are:

- Create a statechart for each natural language scenario. The normal flow, all exceptional flows and all alternatives of a given scenario are captured in one statechart.
- Statecharts are developed and refined along with the scenarios, thus providing for continual validation. As coarse overview scenarios are refined to reflect interactions on task level, new states are introduced in the statecharts, states are expanded to comprise substates, and parallelism may be caught in parallel states, as appropriate.
- A single step in a narrative scenario usually translates into a state or a transition in a statechart. As the steps are mapped to either states or transitions, missing states and transitions will emerge and need to be added.
- Model the normal flow first. Integrate the alternative flows later on. Check if alternatives are missing.
- Represent abstract scenarios as hierarchical statecharts.
- Check the statecharts for internal completeness and consistency. Are all the necessary states specified? Are all the states in a statechart connected? Are states and events named expressively and consistently, following some scheme?
- Check the event list created for scenario elicitation to see if all relevant events are handled.
- Cross-check statecharts. Do states, transitions and events appearing in more than one scenario have the same names?

Creation of narrative scenarios and of statecharts is an iterative process. Statecharts have to be validated with users either by inspection or review, or by paraphrasing sequences of actions in the statecharts in a narrational style. All (important) paths are traversed; the developer guides the customer through the flows. This validation activity works hand in hand with the phase of test case derivation: The paths traversed with the customer to validate the statecharts are test cases that need to be tested in system test.
Representing scenarios with statecharts requires some work to be done that otherwise wouldn't. But the extra work put in scenario formalization pays back in many ways:

1. As mentioned before, the transformation of structured-text scenarios into a semiformal statechart representation helps in verifying and validating narrative scenarios. Omissions, inconsistencies and ambiguities are found. The specification is thus improved.
2. Developers gain a deeper understanding of the domain and the system to be built because they have to understand the details to formalize the scenarios.
3. The statecharts created in the transformation may well be used and reused in design and implementation.
4. The formalized scenarios are (re)used in testing. Test case preparation and expenses are moved from the testing phase late in the development process to earlier activities, thus alleviating the problem of testing poorly done under time pressure. By using a systematical way to develop test cases, test coverage is improved.

The cost of developing the statecharts is justified by the benefits of an improved specification and enhanced testing.

3.2.2 Statechart Annotation

The main problem in using narrative scenarios and derived statecharts as test cases is that they usually are on a level of abstraction that does not allow derivation of concrete test data – that is of input values and expected output – directly.
For this reason we extend the statechart concept to include information important for testing and test case derivation. In particular, the additional testing information included in every statechart should comprise the following:

- Preconditions
- Data: Input, Output, Ranges and
- Nonfunctional requirements

The information is captured in annotations.

In Figure 2 the annotation with preconditions, data items and performance requirements is illustrated.

3.2.3 An Example

As an example of the formalization process, we choose a scenario of the familiar automated teller machine (ATM). Because of space limitations, we only describe the abstract scenario "Authentication" here: A user of the ATM identifying him/herself to the system. For a short specification of the ATM as well as for a more complete example turn to either [12] or [13].

The "Authentication" scenario reads:

1. The customer inserts the card
2. The system checks the card's validity
3. The system displays the "Enter PIN" Dialog
4. The customer enters his PIN
5. The system checks the PIN
6. The system displays the main menu

To keep the description of the "Authentication" scenario short, we model neither alternatives nor exceptions.

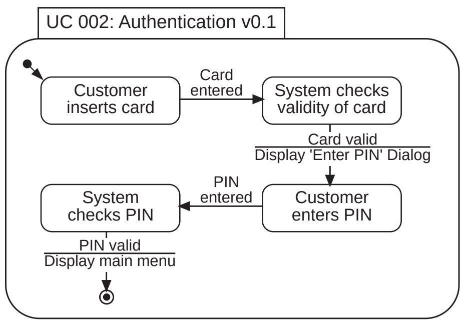
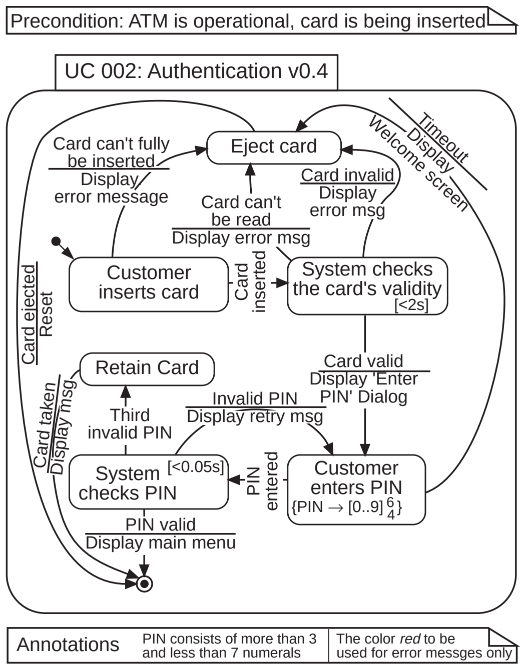


Figure 1: A statechart representing the 'Authentication' scenario



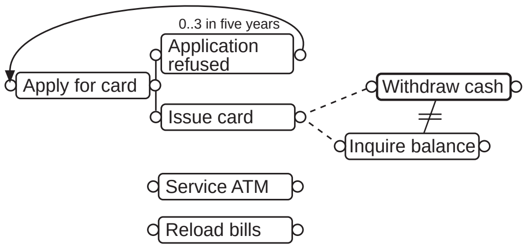


Figure 3: A Dependency Chart for the ATM example

Figure 2: 'Authentication'-Statechart with alternative flows and annotations

Based on the natural language scenario a statechart is developed. At first the normal flow of actions as depicted above is modeled (see Figure 1).
Then the alternative flows are modeled (see Figure 2).
Once a scenario is modeled in a statechart, the statechart is annotated as needed. Preconditions and data are specified. In the example, valid and invalid PINs are distinguished by state transitions, but no indication as to what an invalid PIN is, is made. So a data annotation may specify the range and the form of a PIN (see Figure 2).

3.3 Test Case Derivation

Test cases in the SCENT-Method are developed in a three-step procedure:

1. Test case derivation from statecharts
2. Test case derivation from dependency charts
3. Additional tests (e.g. testing for specified qualities) as suggested by the notes and remarks written down during the analysis phase, during scenario creation and scenario refinement (see section 3.1.1).

Additionally, a fourth step might be performed: The statecharts are integrated and additional test cases are derived from the integrated system statechart.
In this paper we describe the first two steps of this procedure only, because of space limitations. A more in-depth description of the method can be found in [12, 13].

3.3.1 Test Case Derivation from Statecharts

After narrative scenarios have been transformed into statecharts, test cases for system test are generated by path traversal in the statecharts. Any method to traverse all paths in finite state machines according to a given criterion can be used to derive test cases. The method of our choice is simply to cover all links thus reaching a coverage for the state graph comparable to branch coverage C2 in structural testing. A more elaborate coverage could be chosen as desired (e.g. switch or n-switch coverage [1, 4, 10] or comparable methods that consider more than one link at a time). Data annotations enable the tester to easily develop domain tests (boundary analysis, exception testing), performance and timing constraints allow for performance testing. Preconditions specified in the statecharts define test preparation that has to be done before test cases derived from the statechart can be executed – the testing setup is determined by the preconditions.
Path traversal in statecharts will only generate tests for valid sequences of events. For this reason it is important to also include sequences in the test that are not admissible. All events that possibly could occur while in a given state should be tested for.

3.3.2 Test Case Derivation in the Example

To illustrate test case derivation from statecharts, we present in Table 2 some test cases as created by path traversal of the statechart depicted in Figure 2. The first test case follows the normal flow of actions: The card can be inserted and the card as well as the PIN are valid. The next test case considers the exception of an incorrect PIN entered. Next an invalid PIN is entered (PIN too short; this test case takes into account the data annotations specified in the statechart). Finally, a third invalid PIN is entered to provoke another validation failure and traverse the *Third invalid PIN* link.

Table 2: Test Cases

Test preparation:		ATM operational, card and PIN (1234) have been issued, card is being inserted	
ID	*State*	*Input/User actions/ Conditions*	*Expected output*
1.1	Card sensed	Card can be read, card valid, valid PIN (1234) entered in time	Main menu displayed
1.2	Card sensed	Card can be read, card valid, invalid PIN (1245) entered in time (first try)	Retry message displayed
1.3	Retry msg	Invalid PIN (123) entered in time, second try	Retry message
1.4	Retry msg	Invalid PIN (1234567) entered in time, third try	Card retained, user informed
…	…	…	…

3.3.3 Test Case Generation from Dependency Charts

To capture logical and timing dependencies between scenarios, we introduce a new diagram type called *dependency charts*. In dependency charts, scenarios are shown as rectangles with rounded corners and circular connectors. Scenarios without any connecting lines (dependency lines) may be executed in free order, even in parallel if appropriate. Scenarios that have to be preceded by a certain scenario, scenarios that have to start/end before/after an other scenario, sequences of scenarios, alternatives and iterations[1] are shown on dependency charts by dependency lines. A small example is given above to illustrate the representation of scenarios and their relations and dependencies in a dependency chart (Figure 3). A further discussion of dependency charts may be found in [12].
For all the dependencies in a dependency chart, test cases have to be specified. That means, if a scenario has to be preceded by another scenario, try to execute the scenario with and without executing the preceding scenario first. Other dependencies are handled likewise, thus deriving test cases to test the dependencies and interrelations between scenarios.

[1] Alternatives and iterations are seldom used on an inter-scenario level: For this reason the example has been expanded and the scenario "*Apply for Card*" has been subdivided to show an alternative and an iteration.

4 CONCLUSIONS

In this paper we have presented the SCENT-Method, a scenario-based approach to support the tester of a software system in systematically developing test cases. This is done by first eliciting and documenting requirements in natural language scenarios, using a template to structure the scenarios. The narrative scenarios then are formalized using statecharts as a notation. The statecharts are annotated with information important for testing. Finally, test cases are derived by path traversal of statecharts. The method presented in this paper is novel with respect to the synthesis of the aforementioned factors into one single process. We thus supply a method that supports the tester in systematic test case derivation, that uses artifacts of the early phases of the development process in testing again and that handily integrates with existing development methods.

The SCENT-Method has been applied in practice to two projects at ABB in Baden/Switzerland. First experiences are quite promising as the main goal of the method, namely to supply test developers with a practical and systematical way to derive test cases, has been reached. The projects in which the method was applied were applications to remote monitoring of embedded systems [7].

The use of scenarios was perceived by the developers as helpful and valuable in modeling user interaction with the system. On the other hand, scenario management was a major problem throughout the development process.

The formalization process also posed some problems, as the mapping of actions in natural language scenarios to states or transitions is not definite and clear-cut. A narrative scenario transformed into a statechart by one developer may differ significantly from a statechart developed from the same scenario by another developer.

Test case creation was unproblematic as the chosen link coverage in statecharts is simple, yet powerful.

Acknowledgments

The work presented in this paper has been supported by the Swiss Commission for Technology and Innovation of the Swiss Federal Economics Department and by ABB Corporate Research Ltd Baden-Daettwil.

REFERENCES

[1] T. S. Chow: Testing Software Design Modeled by Finite-State Machines; *IEEE Transactions on Software Engineering*, vol. 4, n° 3, pp. 178-187, 1978

[2] D. C. Firesmith: Modeling the Dynamic Behavior of Systems, Mechanisms and Classes with Scenarios; *Report on Object Analysis and Design*, vol. 1, n° 2, pp. 32-36,47, 1994

[3] N. E. Fuchs, U. Schwertel, R. Schwitter: Attempto Controlled English - Not Just Another Logic Specification Language; Logic-Based Program Synthesis and Transformation, *Eighth International Workshop LOPSTR'98*, Manchester, UK, 1999

[4] G. Gonenc: A Method for the Design of Fault-detection Experiments; *IEEE Transactions on Computers*, vol. C-19, pp. 551-558, 1970

[5] D. Harel: Statecharts: A Visual Formalism for Complex Systems; *Science of Computer Programming*, vol. 8, pp. 231-274, 1987

[6] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, C. Chen: Formal Approach to Scenario Analysis; *IEEE Software*, vol. 11, n° 2, pp. 33-41, 1994

[7] R. Itschner, C. Pommerell, M. Rutishauser: GLASS: Remote Monitoring of Embedded Systems in Power Engineering; *IEEE Internet Computing*, vol. 2, n° 3, 1998

[8] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard: *Object Oriented Software Engineering: A Use Case Driven Approach*. Amsterdam: Addison-Wesley, 1992

[9] I. Jacobson: Basic Use Case Modeling; *Report on Object Analysis and Design*, vol. 1, n° 2, pp. 15-19, 1994

[10] S. Pimont, J.C. Rault: A Software Reliability Assessment Based on a Structural Behavioral Analysis of Programs; *Proceedings 2nd International Conference on Software Engineering*, San Francisco, CA, 1976

[11] J. Ryser, S. Berner, M. Glinz: On the State of the Art in Requirements-based Validation and Test of Software; University of Zurich, Institut für Informatik, Zürich, *Berichte des Instituts für Informatik 98.12*, Nov 1998

[12] J. Ryser, M. Glinz: SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test; to appear as a technical report at University of Zurich, Institut für Informatik, Zürich, 1999 www.ifi.unizh.ch/groups/req/ftp/SCENT/SCENT_Method.pdf

[13] J. Ryser, M. Glinz: A Practical Approach to Validating and Testing Software Systems Using Scenarios; *Quality Week Europe '99*, Brussels, 1999

[14] S. Somé, R. Dssouli, J. Vaucher: Toward an Automation of Requirements Engineering using Scenarios; *Journal of Computing and Information*, Special issue: ICCI'96, 8th International Conference of Computing and Information, pp. 1110-1132, 1996

[15] I. Spence, C. Meudec: Generation of Software Tests from Specifications; *SQM'94 Second Conference on Software Quality Management*, Edinburgh, Scotland, UK, 1994