SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test

JOHANNES RYSER MARTIN GLINZ

Institut für Informatik
University of Zurich
Winterthurerstrasse 190
8057 Zurich
Switzerland
{ryser, glinz}@ifi.unizh.ch

Technical Report 2000/03, Institut für Informatik, Universität Zürich.

Abstract

Scenarios (Use cases)[1] – being descriptions of sequences of interactions between two partners, usually between a system and its users – have attracted much attention and gained wide-spread use in requirements and software engineering over the last couple of years. Many software development methods and modeling languages comprise some notion of scenario (e.g. OOSE Object-Oriented Software Engineering/Jacobson/, OMT Object Modeling Technique/Rumbaugh/, and most notably, the UML Unified Modeling Language/Booch, Rumbaugh, Jacobson/ as their successor).

In scenarios, the functionality and behavior of a (software) system is captured in a user-centered perspective. To date, scenarios are mainly used in the requirements elicitation and analysis phase of software development; they are used to capture and document requirements, to enhance communication between the stakeholders (user, procurer, developer, management, …) and to involve the user more actively in specification and validation of the system.

Even though scenarios are mainly used in system analysis, the use of scenarios in other phases of software development is of much interest, as it could help to cut cost by reuse and improved validation and verification. As scenarios form a kind of abstract 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 report, we present a procedure to create scenarios during the analysis phase of system development, to structure them according to a given scenario template, and to use these scenarios in the system testing phase to systematically determine test cases. This is done by formalization of natural language scenarios in statecharts, annotation of statecharts with helpful information for test case creation/generation and by path traversal in the statecharts to determine concrete test cases. Furthermore, dependencies between scenarios are captured and modeled in so-called dependency charts, and test cases are derived from dependency charts to enhance the developed test suites.

The approach has been applied to two projects in industry. In this technical report, we also report on some of the experiences made in applying the approach in practice.

1. In this report, the two terms ‘scenario’ and ‘use case’ are used as synonyms and consequently could be used interchangeably. We will stick with the term scenario though.

Keywords

Scenarios, Use Cases, Scenario-Based Testing, Statecharts in Testing, Scenario Annotations, Dependency Charts

Table of Contents

List of Figures

List of Tables

CHAPTER 1 *Introduction*

The ability to deliver software of high and predictable quality in time, at the same time meeting cost restrictions, is a key competitive advantage for any industry that has a significant amount of software in its products. The timely delivery of complex good-quality software depends on several critical factors. Among them, requirements validation – ensuring that the requirements completely and adequately state the needs of the users – and system test – establishing that the implemented system conforms to its requirements – are prominent ones that are widely recognized as particularly important issues.

On the one hand, requirements validation is important because requirements are a major source of expensive errors [Boe81] that creep into the system early in the development process and often are not caught until system test or – even worse – until the software has been deployed and installed at hundreds of field sites. The number of errors that can be traced to requirements faults range from just a few to over 50% of all errors in an application [Bei90]. Thus, it is of great importance to find faults in requirements early in the software development process through validation activities or to prevent and avoid them by improved methods to elicit, document and validate requirements.

System test, on the other hand, is undoubtedly an important activity as it serves to find errors in the developed system and helps to establish confidence in the reliability and the correct and proper functioning of the system. But it is hard and scrutiny work to create, design and manage system tests.

But despite the fact that requirements validation and system test have been recognized as key elements in developing quality software, yet, at present time, requirements validation on the one hand is often done superficially and unmethodically, if done at all, and testing on the other hand is badly planned for, poorly documented and carried out in a non-systematic way. Furthermore, system analysis and requirements engineering as well as requirements validation is lacking proper user involvement, often because models and documents are not understandable to users. Testing is done under immense time pressure at the end of the development cycle, as test preparation and the development of test cases is done only just before testing starts, even though analysis as well as design would greatly profit from the insight gained by developers in creating test cases and preparing tests.

To improve requirements validation, validation activities have to be an integral part of the development method and user involvement has to be encouraged.

To improve testing in practice, systematic test case development and integration of test development methods with ‘normal’ system development methods are central. Test cases are only developed in a systematic way if clearly defined methods are applied. Test development methods will only be used if they are easy to apply, blend into existing development methods and do not impose an inappropriate over-

head or intolerable cost. A method supporting and guiding the test designer in generating test cases that enable functional end-to-end tests of an application, would be very helpful and much appreciated.

The method introduced in this report is intended to provide support to testers and to improve the state of practice in both, requirements validation as well as system test. It employs scenarios to improve the capturing of requirements and their validation, and makes use of these artifacts of the requirements engineering phase in later phases of the software development process again, specifically in system test.

In the report, we introduce the SCENT-Method – A Method for SCENario-Based Validation and Test of Software. SCENT is – as the name of the method suggests – a scenario-based method to validate and verify requirements (by formalization of natural language scenarios) and to systematically develop test cases (by path traversal in statecharts). We propose the use of scenarios, not solely for requirements elicitation and specification as done in leading object-oriented development methods, but specifically for the development of system test cases. Natural language scenarios are used to capture requirements and get a user-centered description of the system's functions and behavior. Dependencies among scenarios are captured in a special diagram called dependency chart (section 14). Scenarios are validated throughout the creation and refinement process. Natural language scenarios are converted into statecharts, the statecharts being a more formal representation of scenarios. Statecharts in turn are annotated with helpful information for system test. From the annotated statecharts test cases are derived in a systematic manner. Thus we utilize synergies between the phases of system analysis & specification and system test.

Scenarios in SCENT are represented in two ways: as narrative natural language scenarios and in a semi-formal graphical notation. Both of them have their benefits and their drawbacks. Narrative scenarios are easy to create, understandable to customers and users, and do not require special training and education. Users, customers and developers need not learn a new language. But natural language is inherently ambiguous, vague and imprecise. It can be interpreted and builds on common background, knowledge and understanding. If this common ground is missing it will lead to misunderstandings. More formal languages are strong in the capabilities they provide to check system models for completeness and consistency, they do not allow ambiguous and vague expressions. Thus, formalization plays a central role in validating and verifying requirements. By transforming narrative natural language scenarios into a more formal representation, omissions, contradictions and vagueness can be discovered and some of the issues raised by natural language are addressed and solved. On the other hand, formal languages are less understandable, deploy a special language that has to be learned, and furthermore, formal specifications are much harder to create. By using both, a natural language and a formal scenario representation, we try to take advantage of both, at the cost of having to keep two representations consistent. Moreover, by transforming narrative scenarios into a semi-formal representation, the systematic development of test cases and further automation of the validation and testing process is supported.

This report is organized as follows: Chapter 2 serves as a motivational and introductory chapter to the problems of testing. Some approaches to solve the problems are mentioned and our approach – the SCENT-Method – is briefly delineated. In chapter 3, we present the main ideas of scenarios and define the terms used in this article. In chapters 4-7 we present the basic concepts and principles of the SCENT-Method: In chapter 4, we describe the individual steps in the procedure of scenario creation and in chapter 5, we introduce the scenario template used in the method to describe and document scenarios. Furthermore, we introduce dependency charts to graphically capture dependencies between scenarios. In chapter 6, the formalization of scenarios is described, and the integration with existing development methods is sketched. Test case generation from statecharts and dependency charts is described in chapter 7. The report concludes with a chapter on related work, an experience report and some conclusions (chapters 8, 9 and 10, respectively).

The appendices are quite extensive: In appendix A, the scenario template is specified. Appendix B presents a sample scenario description of a simple system using the template. The process of scenario elicitation and scenario creation is illustrated in appendix C. The Derivation of test cases from statecharts and dependency charts is illustrated by example in appendix D. And finally, in appendix E, an overview of the notation used in dependency charts is given.

CHAPTER 2 *Motivation – Testing as a Problem*

Software plays an ever increasing role in nowadays systems in most any application domain – be it in the financial area (banking, insurance), public services, communication, management, embedded control systems or in process control and production processes. To achieve the goals of short time to market and best use of resources – thus meeting target cost and schedule –, a trend to release software that has not been tested properly can be detected. This fact is evidenced by premature releases and (beta) testing done by the customers. Software products – as is the case with other products – differentiate and gain their competitive edge ever increasingly and mainly by price and functionality, and less by high quality [Man99]; the issue at hand being 'All software is faulty, so we might as well take the cheapest'…

This approach might work fine for non-critical applications. But as soon as persons' health or life or the core activities of a company are affected by computer programs, software has to be thoroughly tested. Safety and/or business critical software has to be dependable in all critical functions, because a failure could lead to catastrophic consequences, such as serious injury, loss of life or property or going out of business. The cost of system failure thus is a motivating force to test a system thoroughly, extensively and in a well-planned, systematic fashion.

As a consequence, there has been much research in the area of testing and approaches and methods to support and improve testing proliferate. Yet nowadays testing still is often done in an ad-hoc manner, and test cases are quite often developed in an unstructured, non-systematic way. This is mainly due to the reality of commercial software development (only limited resources are available and only sparse resources are allocated to testing) and less to lack of available methods or lacking problem understanding.

Furthermore, testing is a drudgerous, tedious and wearisome activity which prompts fatigue and inattentive work. Test case development thus is an error-prone task in itself. And there are many more reasons why systematic testing still is not applied to the fullest in many software developing units.

In this section we will take a closer look at some of these reasons and we will foster the idea that, to help overcome the mentioned problems, a rigorous process has to be followed and a sound method to validate and verify the system has to be applied. First we introduce the fundamental terms of verification and validation, then we describe some of the problems encountered in testing, and some of the proposed solutions to these problems. This discussion serves as a starting point to introduce the main concepts of the SCENT-Method.

1 Verification and Validation

In this report, we define validation and verification of software according to the IEEE Standard Glossary of Software Engineering Terminology [IEE90]:

> *Validation*. The process of evaluating a system or component during or at the end of the development process to determine whether it satisfies specified requirements.
>
> *Verification*. The process of evaluating a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase.

In requirements validation, the requirements captured in a specification are tried to see if they adequately and completely state the needs of the users. And in system test the implementation is tested to show conformity to its requirements.

Validation and verification are generally recognized as two vital activities in developing a (software) system. They are especially valuable when applied early in the development process, as errors found during the specification and design phase are much cheaper to correct then errors found in consequent phases [Boe81]. Early validation and verification thus greatly reduce error fixing and fault cost.

System and software validation according to customer requirements is a primary activity during product development, accounting for up to 50% of development cost. Reducing cost in these activities would amount to millions of dollars saved for any big company.

Testing plays an important role in validating and verifying software systems. Despite some advances in recent years and despite some alternative approaches, software testing still is a crucial element in assuring correct functioning of a program and of critical functions. It helps establish confidence in a system's reliability. To do so, it has to be performed in a systematic manner, needs to be based on sound principles and employ appropriate strategies.

2 Testing Problems in Practice

Many strategies and approaches for testing exist. Besides established techniques like control and data flow testing or boundary analysis/domain testing [Bei95, Mye79], formal languages for specification and specialized testing languages are gaining increased attention. Yet testing in practice suffers from many problems. The following enumeration lists the more prominent of these problems:

1. **Lack in planning / time and cost pressure**: In real-world projects, tests are conducted under immense time and cost pressure, as often the project at the end of the development process is late of schedule and over budget already. Detecting faults and detected bugs cause additional delays. As a consequence, both, test preparation and test execution are frequently performed only superficially. Cost and time needed for testing are hard to be estimated with reasonable accuracy. Moreover, testing is often insufficiently planned for and not enough time and resources are allocated for testing.
2. **Lacking (test) documentation**: Tests are not properly prepared, no test plans are developed and tests are not documented [Yam98].
3. **Drudgery, lacking tool support**: Testing and test case development are tedious, wearisome, repetitious, error-prone and time-consuming activities which prompt fatigue and inattentive work, even if sound testing strategies and methods are applied. For this reason, testing has to be supported by tools. But only limited tool support does exist to date. Extended tool support and more especially automatic test case generation is restricted to systems which are formally specified. Even if automatic test case generation may be applied in a formally defined system, the resulting test suites are of immense size and generally only poor coverage is reached.
4. **Formal languages/specific testing languages required:** Some test methods use formal specification languages or specific testing languages (thus requiring special training and education). Their

application is costly, they are difficult to apply and/or can only be applied to limited problems or very specific, restricted domains.

5. **Lacking measures, measurements and data to quantify testing and evaluate test quality**: In most projects only little testing data (error statistics, coverage measurements, and so on) is collected during testing, or available from other projects. Because of missing data, only little can be said about the benefits and economics of testing, different approaches can not be compared and processes can hardly be improved. Often, the quality of tests, and thus – to some extend – of the product, is not assessed. Furthermore, the missing data further aggravates the problem of inaccurate test planning and failed or suboptimal allocation of the necessary resources.

   And finally, there is a problem, which is a consequence of the afore mentioned problems, and a problem in itself:

6. **Insufficient test quality**: Because tests are conducted in an ad-hoc manner and test cases are not developed systematically, the test coverage reached is often very low, in fact it is insufficient to reach even the lowest level that is recommended for practice (branch coverage).

3 Solutions – Some Approaches to Solve the Problem

The issues mentioned in the preceding section may be addressed by various different approaches. In this section, we take a brief look at some of the proposed solutions to testing problems and set forth the key conditions for a testing method to be successful in practice.

There are many different approaches that address one or few facets of the testing problem only. These approaches are valuable in that they help improve a specific facet of testing. Yet, most often, they have severe drawbacks affecting other facets and thus limiting the overall usefulness of the approach. The problems of documentation and planning, for example, may be alleviated by improvements to the testing process and by a list of documents and deliverables that have to be produced during the development of the system, including testing artifacts like test plan, test design documentation, test procedures and test cases. But they impose a strict regimen on the developer and raise development cost. Use of and adherence to appropriate methods, a clear definition of testing criteria and a good mix of testing strategies will help to reach appropriate coverage and improve test quality. But most certainly, development will be more expensive. Moreover, special training and education is needed to use the methods and to learn to define and apply testing criteria properly. Formal languages or specialized testing languages may allow for better automation of the testing process and for better tool support, yet they require extended training and education and are not understandable to untrained customers and end-users. Closer integration of testing with established development methods may reduce cost and the need for special purpose languages, and (re)using software artifacts created in the analysis, specification and design phase may improve efficiency in test design and reduce drudgery and time pressure. Again there is a ‘but’ to it: the integration will impose some more work to be done during analysis and design, and reuse requires the artifacts to be designed and developed with much care and in detail, and then they need to be kept up-to-date.

However, we need to be aware that improved testing will have its price. If we want to develop quality applications, software development cost will increase. But good methods and approaches will limit the rise in cost.

The different approaches to solve the problems in testing may be categorized as follows:

- Test methods and strategies (including all different test case development approaches as well as all formal approaches and special testing languages)
- Test management approaches (taking care of the management, traceability, control and maintenance facet of testing)
- Automation and tool support

In Table 1 some of the perceived problems of testing are listed and some means to help alleviate the problems are indicated. Some of the drawbacks encountered when using the proposed solution(s) are described. These notes are but brief indicators of the drawbacks; there often are many more that could be listed. The drawbacks of using formal languages for example are the loss of understandability to users and customers, the subsequent problems in validation and the cost of applying formal languages because of the special education and training needed. Furthermore, the formalization process is error-prone in itself. Some proposed solutions (as for example the 'Do no testing'-approach or the 'Improved methods' solution) can be applied to almost all problems presented. However, they are marked only on the problems they are most prominently associated with. The proposed solutions are limited in some cases as often they represent no valid solution because they compromise quality or solve one specific problem at the cost of aggravating another. These trade-off decisions are often hard to make.

TABLE 1. Testing problems and proposed solutions

Problem to be solved	**Proposed Solutions**											**Drawback of Proposed Solution(s)**
	Proper (project) planning	Integration with SW devlp.	Improved methods	Systematic test case devlp.	Improved tool support	Formal lang., special test lang.	Measures & measurements	Defined process/procedures	Requ. deliverables/documents	Validation, quality assurance	Education, training	Do no or very limited testing	
Lack in planning	X	x						x					Hard, relies on experience
Time pressure	X	x	x					x				X	No testing --> bad quality
Cost pressure	x	X	x									X	No testing --> bad quality
No documentation	x							x	X	x			Documentation overkill, not project specific
Drudgery			x		X								Tools are no solution in themselves, only means
Missing automation/no automated testing					X	x							Automatic testing not feasible, training/appl. cost
Missing tool support		x			X	x							No automatic testing, training/install./appl. cost
No automated test case generation			x		x	X							Understandability, cost, needed training
Lacking measurements & data					x		X	x					No suitable measures, cost of data collecting, analysis
Lacking measures							X			x			Cost of measurements
Insufficient test quality	x		x	X			x	x		X			Selection of appr. method
No systematic approach/methodology			x	X	x			X					Selection of appropriate approach
Lacking problem understanding											X		Cost
Lacking (process, progress) control							X		X	x			Selection of appropriate measures, collecting data

The meaning of the marks in the table is straightforward: A capital X indicates a solution that aims mainly at solving the mentioned problem, a small x indicates that the proposed solution may contribute

to solve the problem. Obviously, more than one solution may be relevant to one problem, and one solution may be applied to address more than one problem.

As can be seen in Table 1, there are solutions to most of the problems in testing, or rather, there are solutions to every single problem by itself, but an integrated approach to solve all or most of the problems is still missing. In fact, as has been mentioned before, some of the proposed solutions worsen the overall testing problem by solving part of one problem and at the same time amplifying other problems.

In Table 2 the proposed solutions of Table 1 are classified according to the categories given above.

TABLE 2. Categorization of proposed solutions to testing problems

	Proposed Solutions										
	Proper (project) planning	Integration with SW devlp.	Improved methods	Systematic test case devlp.	Improved tool support	Formal lang., special test lang.	Measures & measurements	Defined process/procedures	Requ. deliverables/documents	Validation, quality assurance	Education, training
Test methods and strategies		X	X	X		X		x	x		
Test management	X						X	x	X	X	X
Automation and tools					X	x					

Two of the proposed solutions are especially interesting in the context of this report:

1. Test automation, that is automated, or even automatic testing
2. (Improved) Testing methods

A proposed and valuable approach to solve some of the problems encountered in testing lies in automating testing and – often a prerequisite to automation – in specialized test languages and formal specifications. Formal specification languages, as well as special testing languages, have been around for some time now, and they prove valuable in many projects. Testing tools have been improved and supply vital support for testers.

However, formal specifications and special test languages are expensive to apply, as they require special training and skills. Testing is not a simple task that can be easily automated. 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 [Rys98; Spe94].

Another approach to alleviate testing problems is by improved testing methods, strategies and approaches[1]. But despite an abundance of methods, testing is still a problem in many projects and in most companies. This – in our opinion – is mainly due to the following reasons:

1. **Testing is done in the last phase of the development only**. Developers start the development of test cases only after most of the system development has been done. But testing can (and should) be started with as soon as the specification has been written. By developing test cases early in the devel-

1. For an overview of testing methods and strategies, see [Bei90, Bei95, Kan93, Mye79]. Examples of testing methods and approaches are boundary value/domain testing, control-flow and data-flow testing, user interface testing, state-transition testing, and so on.

opment process, many errors, omissions, inconsistencies and even over-specifications may be found in the analysis or design phase yet. As has been pointed out before, it is cheaper to remove errors in the early phases of development.
That's why the testing method should encourage or even enforce early test case development.

2. **Testing methods are not integrated with (software) development methods**. Testing hardly uses any artifacts of earlier phases directly, but much work is needed to create test cases from the requirements specification and design models, often information that was readily available before. It's easy to leave testing to be done at the end of the development, as testing and test preparation is not enforced earlier by the development methods.
Therefore, a testing method should tightly integrate with the development method that is used in a project.
3. **Test cases are not created/generated in a systematic manner**. Test cases are chosen randomly, by experience, according to some rules of thumb or according to insufficient criteria (statement coverage, input cover, …). Testers are left with no definite procedure on how to derive test cases.
A test method should support the tester to systematically create test cases.

In summary, we deem it most important for a testing method to integrate testing with 'normal' development methods, and to support testers with a procedure to derive test cases systematically. A second key point is that testing has to be taken up early in the development process and not as a last minute effort to show the application to be functional and functioning, much more than to uncover errors and show its compliance to requirements. To achieve this, early test case development has to be supported or even enforced by the development method used, thus alleviating the problem of testing done under enormous time pressure. And thirdly, it is crucial to improve test preparation and test documentation [Yam98].

In this paper, we introduce a new approach to requirements validation and system test to address the issues just described. The approach is based on scenarios/use cases[2] that are created during requirements elicitation and analysis and their formalization to statecharts in subsequent activities. The scenarios and statecharts are used to validate the system to be developed with domain experts. Moreover, the statecharts are used to generate test cases to be utilized in system test.

4 SCENT – An Integrated Approach to Systematic Test Case Development

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 different phases (especially 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 for systematic test case development.

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

2. The terms 'scenario' and 'use case' are used interchangeably in this paper. Use cases are descriptions of a flow of actions, depicting the interaction between user and system. They take a user's view, the system being seen from an external viewpoint and as a black-box. See [Jac92] and the following paragraphs.

The key ideas in our approach are:

- 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 (see chapter 4 for a detailed description of the scenario creation process, in chapter 5 the representation of scenarios is discussed, and in chapter 6 a brief description of the validation activities embedded in the SCENT-Method is given),
- Make dependencies between scenarios explicit and show them in a dependency chart (see section 14 for more details),
- Uncover ambiguities, contradictions, omissions, impreciseness and vagueness in natural language descriptions (as scenarios in SCENT are at first) by formalizing narrative scenarios in statecharts [Har87] (see chapter 6, where we describe the formalization process and define some heuristics to help and guide the developer to transform natural language scenarios into statecharts),
- Annotate the statecharts – where needed and helpful – with pre- and postconditions, data ranges and data values, and non-functional requirements, especially performance requirements, to supply all the information needed for testing and to make the statecharts suitable for the derivation of actual, concrete test cases (see chapter 6 for a description of the procedure of statecharts annotation),
- Systematically derive test cases for system test by traversing paths in the statecharts and in the dependency charts, choosing a testing strategy as appropriate and documenting the test cases (see chapter 7).

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 basic integration of the proposed method in any one of those methodologies is quite simple and straightforward (see section 19 for more details).

In chapter 3, we first establish the notion of scenarios as used in this report and define specific fundamental terms in this context. The use of scenarios in software engineering and the advantages and disadvantages of scenarios (and thus of scenario-based approaches) are discussed. The reasons for taking a scenario-based approach are put forward.

CHAPTER 3 *Scenarios – What They Are and What to Use Them For*

In software engineering scenarios have gained a lot of attention over the last few years as a means to elicit and document requirements. But the use of scenarios by no means is limited to software engineering. Scenario descriptions and scenario approaches have long been used in many fields, as for example in human-computer-interaction (HCI) and strategic planning, to name but those two [Wei98].

In this paper we focus on the use of scenarios in requirements and software engineering (RE/SE). Scenarios play an important role in our approach. But even though scenarios are nowadays ubiquitous, a single, formal, agreed upon definition of what a scenario is, does not exists. For this reason, we specify the meaning of the term scenario as used in this report, and define some related terms.

5 Scenario Definition

In recent years, scenarios have attracted increasing interest as a means to elicit, document and validate requirements. Scenarios are (narrative) descriptions of the use of a system. They are of exemplary nature and are usually recorded in natural language[1]. Often the textual descriptions are enhanced by graphics, diagrams and images. A scenario might be captured by other media as well (video, picture series, sketches and the like).

Scenarios in RE/SE normally capture system functionality as beheld by a (potential) user of the system – the system is modeled as a black-box. This user-centered view is especially apt to capture the behavioral aspect of a system – what happens in response to a user action or an external event, and what results or actions are expected on what input sequences. Scenarios are usually goal-centered in that the way to achieve a goal is depicted [Fir94]. All scenarios of a system together specify the (external) behavior and the functionality of a system.

1. as opposed to formal or programming languages

We define the terms scenario, use case and actor in the SCENT-Method – and as used in this paper – 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 [Jac92] – 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.

Actor – A role played by a user or an external system interacting with the system to be specified.

Abstract scenario – (1) Scenario that factors out common steps used in more than one scenario, (2) Scenario on a high abstraction level, that may be composed of or refined in concrete scenarios, (3) Generic, parametrizable scenario, (4) Type scenario, business process level scenario.

FIGURE 1. Definition of terms

In this report, when we talk about abstract scenarios we mean the first definition: common working steps that are part of more than one scenario are factored out and thus it is much easier to keep scenarios consistent[2].

6 *Why Scenarios: Benefits and Drawbacks of Scenarios*

Using natural-language scenarios to capture, document and validate requirements brings many benefits. But there are some drawbacks as well. In this section we discuss the advantages and disadvantages of using narrative scenarios in software engineering and more especially in development and testing.

First, we take a look at the benefits of natural language scenarios.
The use of narrative scenarios is profitable in that:

- Scenarios improve communication between the different groups involved in system development. In particular they promote, further and support communication between domain experts, customers and software developers. Scenarios are well suited to accommodate communication with the end user as well as with the customer, as they are understandable to most anybody. Users don't need to learn special languages or notations to understand scenarios.
- They help increase the software developers understanding of the application domain: In formulating scenarios by rewording users' statements, the developers of the system acquire a deep(er) understanding of the application domain, its processes and procedures.
- They help in mediating understanding of technical terms as used in the application domain and support the developer in acquiring the vocabulary of the application domain.
- They support the process of negotiating and balancing requirements between different stakeholder(s)/-groups. The requirements specification process may be seen as a process to establish mutual understanding of a system and to balance and combine the demands of all stakeholders. Sce-

2. The term abstract scenario does not refer that much to the abstraction property of the scenarios that are factored out (that is, these scenarios do not leave out important details to focus on the important parts, nor do they abstract from real world, physical properties and concreteness, what normally is meant when talking about abstraction), but the term points to the fact that it doesn't make much sense to execute the steps that were factored out all by themselves. They only make sense in context, that is, embedded in the scenarios they were taken from. The use of the term abstract scenario for describing these factored-out scenarios roots back in the work of Jacobson et al. (see for example Jac92, Jac95b).

narios can perform good services in this process as they capture end-to-end functionality and place requirements in their context.

- Scenarios impose a structure on requirements and thereby structure the application[3]: Requirements are bundled by scenarios in a meaningful way as they are collected and combined according to the workflow they belong to and according to the tasks they specify.
- Scenarios take a user's point of view, not just as a by-product or by chance, but as the driving force to requirements elicitation and documentation. By taking a user-oriented perspective, scenarios promote and advance tight user involvement in the software development process and thus support participative software development.
- Scenarios are a means well suited to capture information about the rationale, the objective and the source of requirements. Intentions and motives for specific requirements, as well as information on changes in requirements, may (and should) be put down in writing. Scenarios facilitate this task quite nicely.
- They help to determine the critical functions in an application and thus to prioritize and package functionality. Furthermore, they support risks estimation in application development.
- Scenarios build a valuable base for user documentation and manuals.

There are also some drawbacks to a scenario approach:

- Scenarios are (in most well-known methods as well as in our approach) natural language descriptions, and thus they are inherently ambiguous and inexact. In the following chapters we will introduce some restrictions to natural language to help minimize the inherent ambiguity. By formalizing scenarios into statecharts (chapter 6), a rigid validation and verification is automatically executed and many ambiguities, inconsistencies and omissions are found. Even though natural language carries these problems, it still pays to use natural language instead of a special-purpose, artificial (formal) language for user-related activities because natural language is readily understandable without any further formal training and without special education.
- Scenarios give only a partial view of a system. They do not comprise qualities and non-functional requirements. Furthermore, they do not readily provide 'the full picture', that is, they are not an integrated model of the full system, but rather describe part-functionalities as perceived by the user. Moreover, relationships between scenarios are not modeled in most scenario approaches.

Some minor drawbacks (and solutions to overcome them) are:

- Scenarios are well-suited to depict a system in black-box view and in a user's perspective. They are not appropriate for capturing internal processing and calculation. For these purposes a mathematical formula, a technical description or some graphical model is much more apt and appropriate. However, all of these may be integrated into scenarios as appropriate.
- Natural language scenarios are not apt to describe graphical information: Masks and layout or arrangement of elements in user interfaces are hard and complicated to describe in natural language. Again it is far better to enrich scenarios with pictures, screen-shots and mask designs of screen representations and dialogs than to capture these kinds of requirements in natural language.
- Another shortcoming of scenarios related to the one just mentioned is, that scenarios are not apt to capture requirements that are valid system wide. Requirements that affect many tasks and components of a system and consequently show up in most scenarios are better captured only once in a central repository, to ensure consistency, changeability and maintainability. In this category fall almost all qualities and most non-functional requirements, as for example usability requirements, look-and-feel standards and some performance requirements.

3. This does not mean that design and implementation will or should follow this structure! Scenarios partition a system or a system's functionality in a transactional way. An object model on the other hand will structure a system according to unity of data and operations on that data.

Finally we want to point to the fact that scenarios need to be managed, versioned and maintained like any other artifact of the software development process. This seems quite obvious, but according to our experience it is one of the major problems encountered in working with scenarios. Consistency is one problem. To keep scenarios accurate and up-to-date is a challenging task. Changes in requirements as well as changes in system design and implementation need to be propagated and reflected in the scenarios. Tools support is needed. But there are only few tools available to support a scenario life cycle and scenario management. Most work still is done in word processors and graphical packages. Traceability is another problem necessitating requirements to be traceable through all the different models and representations from analysis to testing. Single requirements in the specification as well as part-solutions in the design, modeling elements in graphical models as well as code fragments in the application-code need to be uniquely identified and interconnected.

In summary, scenarios have the advantage of:

- being understandable to every- and anybody, thus supporting communication between stakeholders
- being natural to human thinking and human problem solving, as humans perceive most activities as a sequence of cause and effect or stimuli/responses[4]
- describing a system in a dynamic view, capturing behavior and emphasizing the importance of an end-to-end view of a whole process, yet helping to find alternative flows and exceptions to normal behavior

The main advantage of scenarios is at the same time their main weakness: Scenarios in their simplest form are informal text flows written in unrestricted natural language. They are hard to manage and hard to be supported by tools, as unrestricted natural text can not be handled automatically. Even though scenarios are abstract test cases for the system under development, it is not easy to use scenarios in testing or to systematically develop test cases from scenarios. Natural language scenarios are hard to analyze, it is hard to keep them consistent and up-to-date, and it is hard to prove their completeness. Ambiguity and vagueness are further problems that need to be solved.

To make further use of scenarios (automation, tool support), they would have to be formalized. But by formalizing them many of the advantages of scenarios would be lost. More especially, the understandability to users and customers suffers. Thus, the justification for the very being of scenarios would be denied. That's why some authors argue that scenarios should not or must not be formalized [Jac95a, Jac95b].

7 Use of Scenarios

In a survey on selected industrial projects [Arn98], we found that scenarios are mainly used to document and validate requirements[5]. Hardly any project or well-know method makes any further use of scenarios created at the time of requirements elicitation and system specification. Yet scenarios can be very useful in testing, as they define abstract test cases on a system level.

Why then are (narrative natural language text) scenarios not used in test case generation for system test?

There are many reasons, the most eminent being:

4. Scenarios take up this notion by modeling the interactions between a system and its environment as a sequence of events and actions.
5. We will not take up any further findings of the survey in this report. However, for anybody interested in an overview of the use of scenarios in practice, we refer to the publication [Arn98].

1. Scenarios aren't formal

   Scenarios usually are natural language text. As they are not formal (often not even structured), it is difficult to use scenarios to generate test cases as they do not provide for a structured, systematic way of test case creation and as test case generation from natural language scenarios can not be automated. That is why in our approach we formalize scenarios in statecharts to use them as a basis for test case generation.

2. Scenarios need to be kept up to date

   Scenarios need to be versioned, managed and kept up to date as does any other analysis model, (design) document or the source code. Yet in most projects this demand is not perceived or not fulfilled. Changes in requirements, faults found during validation activities and changes during design and implementation need to be back-propagated to the scenarios and the specification. Scenarios need to reflect the true requirements, the latest decisions and the actual system being built at all times. This requires a formal scenario management process[6].

3. Scenarios are seen as a supporting means only, but not as software engineering artifacts

   In many approaches narrative scenarios are used but as a tool to support the building of 'real' models (e.g. class models, state automata, …). They are not recognized as system models and part of the specification of their own right. Thus, the misconception is that they do not need to be kept up to date, put under version control, configuration and change management, much less even than to be considered a part of the system specification document.

4. Scenarios are perceived as a requirements engineering model only

   Often scenarios are taken to be of use in requirements definition and specification only. In our opinion, scenarios may be useful in other respects as well, one of them being system test. In fact, some authors emphasize that scenarios are essential to any project [Jac92]. In every project, so they argue, scenarios (or use cases) have to be developed in one way or another, as user documentation and system test rely on a user-centered system view. It is profitable and advantageous to develop the scenarios early in the development process and (re)use them in other activities as well. There has to be a supporting method though to make use of scenarios in testing.

5. Scenarios are but partial views of a system and (usually) at quite an abstract level, they are a high level description of the system

   As scenarios are modeling the system in a user's perspective and as a black-box, they are not well-suited to be used in unit or component tests, which heavily rely on white-box testing. But they might well be a good help in system test, for that is the level on which they capture requirements and describe the system.

The limitation of scenario use to the analysis phase is unnecessary and restricts the usefulness of scenario approaches. It is our goal to define and establish a method to use requirement scenarios for the validation of requirements and in system test in addition to their use in the specification phase where they are used to elicit and document (user) requirements (see Figure 2). Furthermore, we aim at including non-functional requirements, especially performance requirements, in scenarios as well.

We prefer a scenario approach over other approaches as scenarios are especially well suited to actively involve the users in the development process. This is mainly because scenarios have a a simple structure, they are simple to use and understand. Thus, scenarios are especially well suited for validation activities with the user.

Furthermore, scenarios can be – and have been – integrated into most development methods. The specification process does not have to be changed much, only minor adjustment is needed to adapt an existing specification process to scenario use.

6. It is not a scenario management process that is needed, really, but a process to manage any artifacts of the software engineering process, enforcing proper change propagation and requirements traceability. But too often, even if such a management process exists, scenarios are not taken to be a part of the specification, they are not considered to be system engineering artifacts and thus are not subjected to this process.

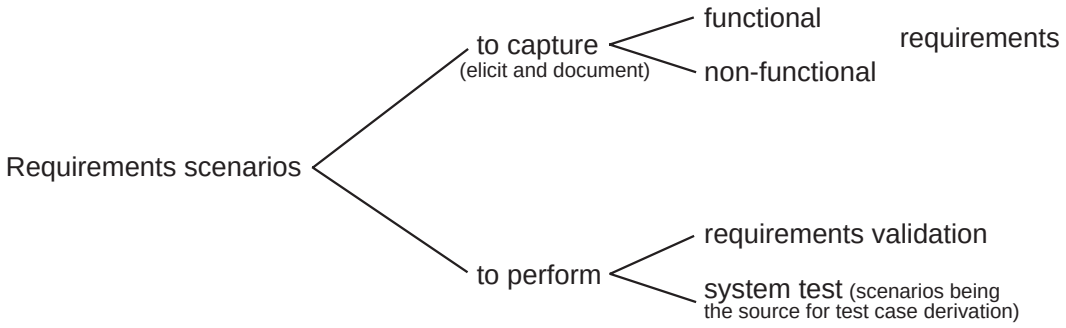


FIGURE 2. Use of scenarios

CHAPTER 4 *Scenario Creation*

Since Jacobson published the book “Object Oriented Software Engineering: A Use Case Driven Approach” [Jac92] in 1992, much work has been done on scenarios/use cases[1]. Yet, most scenario approaches lack a comprehensive, definite scenario creation procedure, a cookbook on how to elicit scenarios from users, how to document and refine them, and finally how and when to validate them.

In the following section a process to create scenarios is introduced. It is important to realize that parts of this process are intended to be applied iteratively and will be run through (at least in parts) many times in a real project.

8 Scenario Elicitation

In this section, we describe briefly the creation of short, abstract-level, natural language scenarios, by specifying the steps that have to be taken to determine scenarios. In the following section, the refinement of high-level scenarios down to detailed descriptions of event-action-sequences is described.

In a first phase of interlaced, mutually dependent activities (steps 1-3), we identify actors, who interact with the system, external events, the system has to react to, and results, the system has to produce. Both, external events and results, also help identify actors we might miss otherwise.

1. Create a list of all persons who work with, use or interact with the system. Determine and define the roles these persons play. All roles are listed. They define the actors.

   Ask questions like: Who is/will be working with the system? Be careful/mindful of different roles a certain person might hold, each role is represented by a specific, single actor. Consider the difference between ‘Who is working with’ as opposed to ‘Who is operating, administering the system’.
   What other systems interact with the system under consideration? Are there any actuators, any sensors? Systems that interact with the system to be built, as well as actuators and sensors are considered to be actors of their own right. Does time play a role (automatic tasks triggered by time events and the like)? If so, time may be modeled as an actor, too.

1. See for example [Ben92], [Hsi94], [Fir94], [Pot94], [Reg96], [Coc97], [Sut98], and [Hol90], the last mentioned article being one of the precursors of modern scenario-based methods that introduced the use of scenarios in software engineering. [Fil98] gives an overview of and compares the features of the most prominent scenario-based approaches. Some more detail about scenario approaches related to the SCENT-Method may be found in chapter 8 of this report.

Actors may be arranged in an inheritance hierarchy if desired and appropriate, that is, if it helps to make the model more clear. This might be the case, if a large system shall be depicted on different levels (e.g. when on first level users just fall into one category, and are shown as general users on the top level overview diagram only. On a refined diagram, the developers might want to distinguish normal, power and priority users. The general user on top level then shows all associations that are either normal, power or priority user associations. This construct is provided to facilitate hierarchical decomposition for actors also, as scenarios may be decomposed to support analysis and modeling of big systems. Decomposition of scenarios is addressed in section 14).

Results: - List of actors

2. Determine all (system relevant) events: What system external events can occur? Who is triggering these events? Identify actors by determining the events that trigger (re)actions in the system. On the other hand, questions like 'How does every actor work with the system?' will help identify associated events triggered by the actors. Create a list of all events.

   Results: - List of system external events causing the system to (re)act, triggering events
   - Extended list of actors (from step 1)

3. Determine the results achieved by or expected of the system, and the outputs the system produces, as well as the inputs it needs and it receives, respectively. List inputs and outputs in separate lists. What outputs are created, who is the output directed to, to whom are the results given, to whom are dialogs and screen output displayed? Questions like these help identify further actors. Identify all inputs to the system. What data is needed to create every output? Who or what is generating the inputs? Be aware of overlapping between input and (triggering) events.

   Results: - List of outputs
   - List of system inputs (overlaps in parts with system external events list created in step 2)
   - Extended list of actors (from step 2)
   - Actor-Input-Output-Matrix (if desired): Which actor creates the input or receives the output, respectively. Actors constitute the columns, in- and outputs the rows.
   - Input-Output-Matrix (if desired): What input is needed to generate/calculate what output?

Once the actors have been identified, we determine the boundaries of the system. Then we create first high-level scenarios, which consecutively will be refined.

4. Determine system boundaries. The steps mentioned so far help to demarcate and delimit the system. They help identify the borders of the system: What is part of the system, what lies outside the system? What can be influenced or even determined by the system? Actors are external to the system. Time may be considered to be an actor.

   Results: - System boundaries, context diagram (e.g. an overview diagram without use cases yet)

5. Create high-level scenarios just featuring a name and a short description (two to four sentences): How does every actor work/interact with the system? How does the system or how do the actors react to every single event of the event list? What invariants and restrictions do exist? Scenarios may be at the type or at the instance level, that is, scenarios may describe interactions from the point of view of a given individual (Fred Myers does …) or in a more abstract view (the system administrator does …). On this level of abstraction, scenarios are on business process or task level.

   Scenarios and actors may also be found via system functionality: Who needs a certain functionality, who will be using it (actors), for what is this function used (scenario), who (what actors and what other systems) has part in this functionality?

   What are the results of an operation? How are they obtained? Whom are they given to?

   Who or what is triggering a given function (actor or event)? Are there any operations that need to follow a given operation?

   Give each scenario a descriptive name using short active verb phrases and avoiding similar names.

   Results: - List of preliminary scenarios (the ovals in an overview diagram), scenarios are named and carry a short description

Steps 1 to 4 need not be executed in any specific order, no sequence is implied. They are interlaced and interrelated activities helping to identify actors and to check the scenarios for completeness in later activities.

A glossary of terms has to be created. All actors, events, scenarios and activities, as well as all data items, in- and output and conditions, if necessary, are named, defined and shortly described. Synonyms are listed and dependencies between different items are specified.

Scenarios may be described in very different styles and form. To categorize the different scenario types and define the style scenarios are to be converted to in SCENT, we characterize a scenario according to its level of abstraction in different aspects. Abstraction has many facets. We use the three dimensions of actor type abstraction, granularity and scope: scenarios are on type or instance level with respect to the actor type abstraction, they are on business process, task or working step level with respect to their granularity, and they describe system internal behavior and sequences of actions, interaction of the system with users and other systems, or they depict associations, relations and interactions between actors, system and environment, capturing contextual information also [Arn98]. The three dimensions, and the values that belong to them, are:

- Actor type abstraction: instance vs. type
- Granularity: working step vs. task vs. business process
- Scope: internal vs. interaction vs. context

Up to now, scenarios may be instance or type scenarios (e.g. Jim Kirk does … vs. The captain does …), they are but on the level of business processes or at most tasks, and they feature no internal and hardly any interaction information. In the end, we want to have type scenarios that define all interactions and possibly internal working steps.

Now that coarse initial scenarios have been created, the scenarios are refined in a process that is described in the following paragraphs.

9 Scenario Creation and Refinement

6. Define all scenarios and prioritize them according to their importance. The scenarios in step 5 may be incomplete, overlapping and without any order. The intent in step 5 is but to create a list of the main scenarios of a system. In this step we focus on covering the whole functionality of the system and on packaging scenarios into releases by assigning priorities. After the main scenarios have been defined in step 5, add less important scenarios (described just by title, goal and purpose yet). Cover all functions of the system. Are scenarios complete now, that is: Is every functionality of the system covered by at least one scenario?[2]

Scenarios are still on the business process or task level, single working steps are described in the next step only. Usually, the scenarios will be type, and not instance scenarios (that is, scenarios

2. What is/will be the use, the purpose of the system? What shall the system be able to do? What tasks are to be handled by the system? How will users work with the system (expected use and wishful thinking)? Who else is working with the system? What functionality is needed? What information and what data is to be handled and managed, to be stored, to be created, ...? What results are to be calculated, generated, what output is created, ...? What kind of system is to be developed? Is it a new system, a replacement of an existing computerized system or a replacement of a manual system, is it a prototypical development, is it a system in an established domain where a deep understanding of the domain and of the application as well as much experience is available and problems have been solved before or is it a newly evolving domain? Is it a one-of-a-kind development or a “standard application”, a development for the market or for one specific customer, ... Use existing specifications, similar systems and existing know-how/the experience of developers who have developed systems in this domain before.

depict the flow of actions not from the point of view of a given individual, but from a more abstract view).

Define the scenarios in dialog with the user(s), not describing the detailed flow of actions yet. The goal is to find all (relevant) scenarios to cover all the functions and uses of the system (completeness). Check that all the events of the event list are taken care of, that is, that they are handled in or are triggering at least one scenario. Make sure, all the inputs to the system and all the outputs from the system are accounted for, that is: Are all the inputs processed or handled in at least one scenario, are all the outputs produced in at least one scenario?

This is the first validation activity in our procedure. It helps check that the scenarios cover all system functionality and it calls for and promotes an early feedback from users. In this step the user(s) should notice if user scenarios are missing. The cooperation and dependencies between scenarios should be documented and checked (see section 14).

Order scenarios by importance. What scenarios are most important in system context, which scenarios describe core functionality? Prioritize the scenarios in a joint session with the customer (user and procurer) and the developers (technical estimates, feasibility, implementation cost, ...)

Results: - List of scenarios marked with priority information

- Links scenarios – actors (e.g. as a table with the actors being the columns and the scenarios defining the rows)

7. Create a description for each scenario using a template to facilitate a common structure, and restricting the refinement to the scenarios of highest priority at first. Describe the (coarse) flow of actions in each scenario, define a step-by-step procedure of what is typically sequentially done in working with the system to achieve a given task or goal. Furthermore the scenarios are to be named, ordered and structured. Instance scenarios are aggregated to type scenarios.

Rework names. Are the names summarizing the goals of the scenario? Are they aptly capturing a scenario's purpose? Apply a numbering scheme to the scenarios. Define the numbering scheme to reflect the priorities assigned in step 6. Is there an order to the scenarios, can scenarios be grouped? Which actor triggers which scenarios? What scenarios are functionally related? Is a given scenario similar to other scenarios? Can similar tasks or working steps be identified and factored out in abstract scenarios? Can scenarios be merged? There is no need to extract all abstract scenarios at this time. In step 12 this will be the main focus. Here it suffices to identify similarities in scenarios and arrange them accordingly (e.g. numbering scheme, naming scheme, etc.)

Results: - Flow of actions in scenarios

8. Draw a dependency chart showing all dependencies between scenarios (section 14). Create an overview diagram – based on the scenarios determined in the preceding steps – to show the triggering-relations between actors and scenarios (section 13).

Results: - Dependency chart and overview diagram

9. Have users review and comment on the scenarios, the dependency chart and the overview diagram. This is a second validation step which can be quite informal. Have users look through and annotate the scenarios. Incorporate enhancements and remove errors found by the reviewers.

Results: - Findings, comments and errors annotated to scenarios

10. Refinement of scenarios: Extend the scenarios by refining the description of the normal flow of actions down to the desired level of abstraction[3]. Integrate the improvements and correct the errors found in step 9. Specify normal behavior down to single working steps, describe the sequence of stimuli and (systems) responses in detail. The task-level scenarios are refined into scenarios containing "atomic" working steps. Annotate scenarios with abstract test cases and with information helpful to derive test cases later on (e.g. hints what to mind and what to be aware of, what to test for especially and the like). This step might not be necessary depending on the work done in the preceding steps. Yet it proves valuable to take the time to annotate scenarios with comments concerning the testing to be done.

3. For example: shall in a scenario description user input be broken down to single key strokes, or is the action of 'entering a user-name' to be considered as a single, atomic action?

Results: - Description of the normal flow of actions of every scenario
- Hints, instructions and information for test case derivation

11. Model alternative flows of actions, specify exceptions and how to react to exceptions. Ask questions as to what will happen if the user does this or that, what will happen if now this or that event or interruption occurs, what are alternative actions that can be taken, under what circumstances would the user have to (re-)act differently, are there different classes or categories of input that the user or the system has to react differently to, what conditions lead to different behavior, what could go wrong in executing this task, what could go wrong in each working step, ...

Annotate scenarios with abstract test cases and with information helpful to derive test cases later on (e.g. hints what to mind and what to be aware of, what to test especially and the like)

Results: - Description of alternative flows of actions, of exceptions and their handling in scenarios
- Hints, instructions and information for test case derivation

12. Rework the scenarios to find common working steps. Factor out like sequences into abstract scenarios to be used in several normal scenarios thus diminishing the expense to keep scenarios consistent. Overlapping scenarios are to be identified and similar behavior is to be factored out.

Results: - Abstract scenarios (extended scenarios as needed)

13. Include non-functional requirements (performance requirements, user interface description, ...) in scenarios.

Results: - Scenarios annotated with non-functional requirements

14. Revise and adjust the dependency chart and the overview diagram based on the refined scenarios, including abstract scenarios and alternative flows.

Results: - extended, revised dependency charts and overview diagram

15. Have users check and validate the scenarios. Enforce whenever possible a formal review cycle. Iterate through all or parts of the steps above (10-15) as often as necessary.

Results: - Validated scenarios, after last iteration cycle: “final” version[4] of scenarios

16. Structure the scenarios according to a given template. Make sure all the needed information has been gathered. Specify qualities and non-functional requirements and annotate the scenarios accordingly, that is, if a quality affects but one (or few) scenario(s) document the quality requirements in the scenario descriptions. Fill in additional information as required, ask for missing information. Annotate scenarios with abstract test cases and with information helpful to derive test cases later on. The final product of the scenario creation procedure is a specification paper consisting of

- Dependency charts, overview diagram (if desired)
- List of actors and list of scenarios
- Scenarios structured according to a given template, listing the purpose of the scenario, a short description of the scenario, the normal flow of actions, all alternatives and exceptional flows, the participating actors, pre- and post conditions and the triggering event
- Hints, instructions and information for test case derivation and extended testing of the scenarios

While the task of structuring the scenarios according to a given template is presented here as step 16, as if it were a single action that is executed at the end of the process, it has to be understood that the scenarios are structured and organized according to the pattern specified in the scenario template all along the way[5]. The final activity done in step 16 is merely for checking that the information needed to make full

4. The term “final” as used here in this context is to be taken as “approved”, it does not mean that scenarios are frozen and should not be changed and adapted to changing requirements or changing system environment, but it marks a version of the scenarios that the customer and the developers have agreed on to serve as the base for development. Changes to requirements after this point are only integrated if approved by the stakeholders and applying strict change management and version control.

5. In fact, the scenario description template is first introduced in step 7 – when a step-by-step description of the scenarios is created – to help structure the scenarios and to record additional information, as for example an owner (of the scenario), a version number, and pre- and postconditions.

use of the scenarios is complete and documented, and that scenarios are structured according to the template. Furthermore, an active, conscious effort is undertaken to capture information that is relevant and helpful for testing and test case derivation.

The full fledged process as described here may be tailored to meet project needs. The tailoring is quite easily done: prioritizing can be skipped if not needed, the scenario description and refinement may be done in two or even one step, only, instead of three steps (steps 5, 7, 10), alternatives (step 11) may be specified in the course of describing the normal flow, and also the factoring out of abstract scenarios may not be needed, depending on problem size.

In the following paragraph we give some brief advice on how the procedure is to be applied. More especially, the question of who is doing what, and how to group the steps, is shortly discussed. This information is graphically represented in Figure 3.

Steps 1 to 6, are an integral, strongly coupled package with many reciprocal effects, interdependencies and interactions, and thus, they are best done in one meeting. Analysts, RE/SE engineers, developers, customer(s), end-users and other stakeholders (depending on the organization, you will include testers, maintenance and quality personnel, management/project management, system administrators and domain experts) participate in this first joint session. The results of this first meeting, often being but notes and sketches, are analyzed and documented in step 7. An overview of the system as depicted by scenarios is created in step 8. These last two steps are usually done by the analysts/developers, but require close user involvement to resolve questions arising in describing and refining the scenarios. The scenarios are then reviewed by the customer, end users and other domain experts in step 9[6].
Steps 10 to 15 comprise error correction, refining the model and validating the scenarios in an iterative process. The iterating process may be started in another joint session during which alternatives and exceptions are specified. Or developers try to identify the alternatives and exceptions themselves – if they have the needed domain knowledge – and then, they have them reviewed by the customer/users. Close cooperation is required between developers and customer during these iterating steps to ensure that the scenarios are capturing true requirements as demanded by the users. Yet, most of the activities in this bundle are up to the developer, except validating the scenarios which requires active user involvement.

For an example of a template to structure scenarios see appendix A; an exemplary scenario description can be found in appendix B. Figure 3 shows the scenario creation process graphically, shading depicts user involvement (the heavier the shading, the more user involvement is required).

A concise tabular representation of the steps in the scenario creation process is given in Table 3. Some hints as to the integration of the scenario formalization procedure (see chapter 6) are included in the tabular description of the scenario creation procedure as remarks in square brackets.

6. This activity may be supported by a checklist for reviewing scenarios.

TABLE 3. Scenario elicitation, scenario creation and structuring

#	Step Description	Results
1	Find all actors interacting with the system	List of actors
2	Find all (relevant system external) events	List of events (triggers)
3	Determine results and system in- and output	System in- & outputs
4	Determine system boundaries	Context diagram
5	Create coarse overview scenarios (instance or type scenarios on business process or task level), name, short description [First high level statecharts are created: Few states, no fully specified behavior]	List of scenarios
6	Prioritize scenarios according to their importance and assure that the scenarios cover all system functionality (completeness)	List of prioritized scenarios Links scenarios – actors
7	Transform instance to type scenarios. Create a step-by-step description of events and actions for each scenario (task level), structuring scenarios according to a given template [Statecharts created in step 5 are modified and extended to describe fundamental behavior]	Flow of actions in scenarios
8	Create a dependency chart and an overview diagram	Dependency chart, overview diagram
9	Have users review and comment on the scenarios and diagrams	Comments to scenarios
10	Extend the scenarios by refining the description of the normal flow of actions, break down tasks to single working steps [Statecharts are refined along with scenarios]	Description of normal flow of actions Hints on test case derivation
11	Model alternative flows of actions, specify exceptions and how to react to exceptions	Alternative flows of actions, exceptions and their handling in scenarios Hints on test case derivation
12	Factor out abstract scenarios	Abstract scenarios
13	Include qualities, non-functional and performance requirements in scenarios	Scenarios with non-functional requirements
14	Revise the dependency charts and overview diagram	Revised diagrams
15	Have users check and validate the scenarios (Formal reviews)	Validated scenarios
16	Structure scenarios according to given template; create preliminary test cases / write test plan and specification	Scenario specification paper

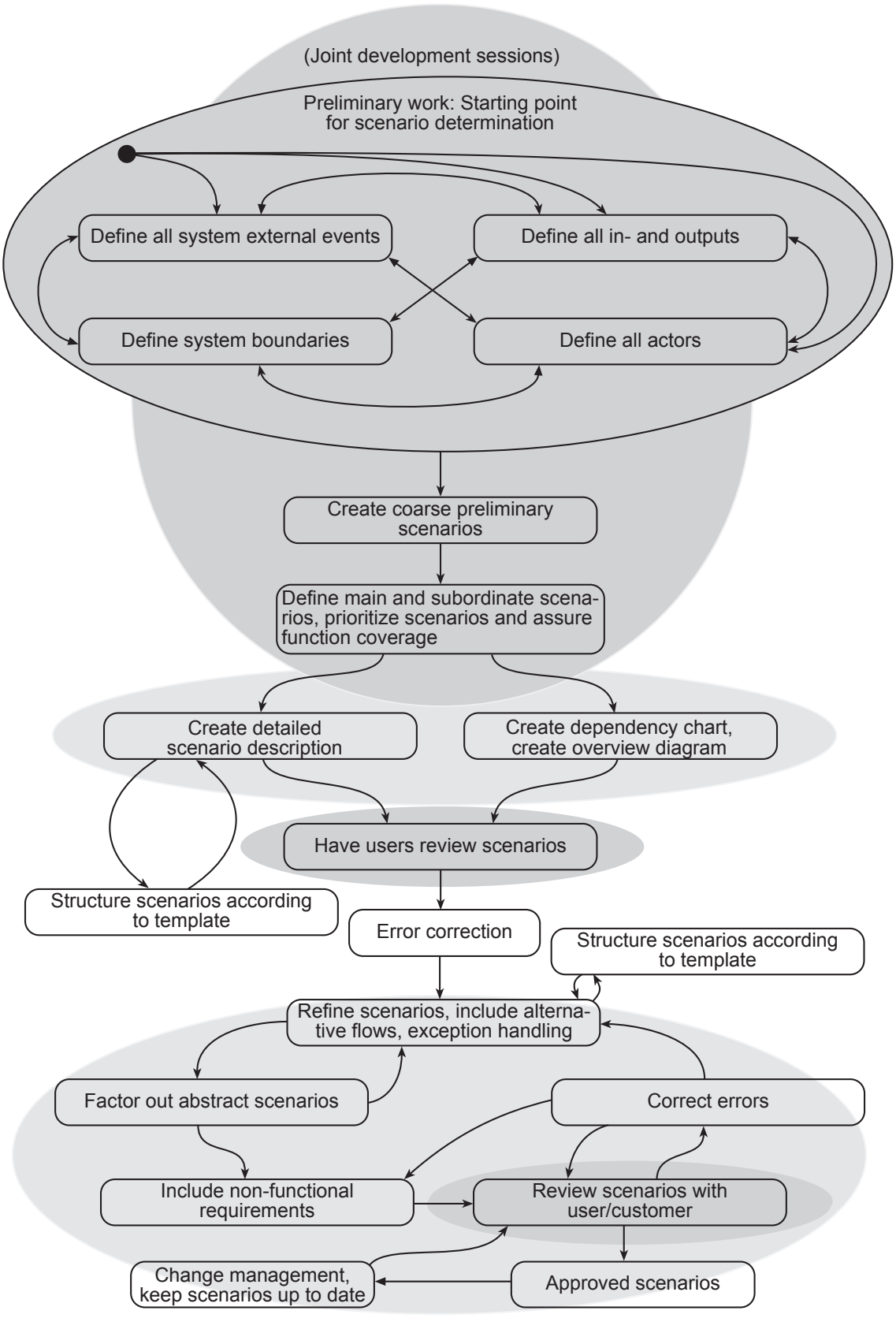


FIGURE 3. Scenario elicitation, scenario creation and structuring

CHAPTER 5 *Scenario Description*

10 Representation Forms

How do we describe scenarios? What information should be documented? What form do we choose to represent scenarios?

There are mainly three possible representation forms or any combination of them, suited to document scenarios:

1. Structured natural language (e.g. using a template)
2. A tabular representation
3. A graphical representation

All of them may be enhanced by further means as for example pictures, sceenshots, user interface masks and the like. As a sidebar we remark that an orthogonal dimension is formality. All of the mentioned representations may be more or less formal. Natural language descriptions are usually informal, but they may be turned semiformal, if keywords (process words, data items and the like) are defined in detail thus assigning a specific semantic to the meaning of a sentence. Tabular representations may be informal – just using unrestricted natural language –, or they may be semiformal or formal as well – using a restricted form of natural language or a formal language. The same is true for graphical representations, which mostly are semiformal, but might be given formal semantics. The dimension of formality and more especially the advantages and disadvantages of formal, semiformal and informal models will not be discussed any further in this report. However, in chapter 3, we list some of the benefits and drawbacks of using natural language (=informal) scenarios.

In SCENT, we use unrestricted natural language to capture requirements and specify the system's behavior and functionality in narrative scenarios, thus taking a fully informal approach at first. Then scenarios are formalized in statecharts – a semiformal representation of the scenarios.

What are the benefits and drawbacks of the respective representation forms? Should a graphical representation be preferred for the task at hand? Or is it beneficial to use natural language? In the following paragraphs we briefly discuss the advantages and disadvantages of the mentioned representation forms.

A graphical representation seems enticing at first, as human perception favors graphically presented information, that is: more information can be conveyed in shorter time by a graphic, a picture or an image than by means of written word or numbers ('a picture says more than a thousand words'[1]). Yet there is some stringent limit to it. As soon as the number of graphical elements[2] surpasses say 10 to 12

elements, the picture is getting overloaded and cluttered. It's getting harder to retrieve the information captured in the diagram.

Another drawback is that graphical notations often are not precise as the exact meaning (semantics) of all graphical elements has not been properly defined[3] or – if defined – the language has to be learned. We don't want the user/customer to have to learn a new language, or many of the benefits of scenarios will be lost. But couldn't we construct a graphical language so intuitive that anybody could understand it without any training? Probably so, but still the exact semantics of the graphical elements would have to be defined and learned and the language would of necessity be that general that most information in such a diagram would have to be captured by natural language names and annotations. So we use diagrams where they are appropriate, namely to give an overview picture of a system, showing all the scenarios of the system, and to capture and picture dependencies and interrelations between the scenarios.

In scenarios, we mainly have to represent sequences (scenarios are by definition – Figure 1 on page 12 – sequences of actions). That can conveniently be done in natural language using numbering schemes, bullets and indents.

Thus, graphical notations, while important and useful, aren't sufficient as rationale, decisions, delicacies and subtleties go undocumented and non-functional (performance, ...) as well as functional requirements not captured by the graphical representation have to be documented in annotations and naming. These annotations that have to be added to specify and document the missing elements clutter the picture and diminish the clearness and intelligibility of the visual representation.

A textual representation on the other hand carries all the problems of natural language, if no restricted, precise language is used. Besides the problems of ambiguity and vagueness that have been discussed before (section 6), natural language is less suited for showing the big picture and helping the user get an overview of the system. It is less apt for showing clearly the connections, relations and (inter-) dependencies between structures and elements. Thus, structures and relations are much harder to grasp and understand than in a graphical representation. However, natural language is understood and spoken by everybody without further special training and education, and it is easy and simple to use. These advantages are invaluable in the analysis phase as they enable users and customers to be involved in the development process without learning special notations and languages. Thus the use of textual natural language scenarios helps not to distract users and customers from accomplishing their real task (e.g. domain expertise, requirements definition and validation).

As to a tabular or a flowtext representation: The choice relies on personal preference, the scenario template presented in the following section can easily be transformed to a tabular form and vice versa (as has been proposed by A. Cockburn [Coc96]).

In a first phase where user involvement is very important and understandability is key for validation purposes, it seems a reasonable and profitable choice to use natural language to describe scenarios. In a second phase the advantages of a semiformal, graphical notation are important. For scenario validation and in testing, we need a representation that easily can be checked for consistency, completeness and that prevents vagueness and ambiguity. A graphical representation with a formal semantic is a big help in

1. But numbers and words often are more precise. Graphic representations are suited to convey proportions and ratios as well as flows, dependencies and relations. Text is best suited to capture exact and detailed information.
2. In fact it's not the element count that is important, but the number of concepts and the number of semantic carrying structures. If a point, a line, and so on is considered a graphical element, then a meaningful structure may well be composed of a dozen (or thousands, for that) graphical elements, and yet be conceived as one structure, one logical element (e.g. an icon, a symbol, a pictogram). It is the count of structures and of concepts, only, that counts.
3. Not mentioning the problem that many graphical elements are heavily overloaded and have different meanings in different methods. This leads quite often to confusion, misunderstandings and frustrations as yet another notation has to be learned, yet another meaning is given to a graphical symbol and yet another interpretation of a diagram is possible depending on the variant and version of the method used.

achieving these goals. Test case derivation is systematized and even can be automated by using state machines to represent scenarios.

For all the mentioned reasons we choose to use both representation forms, and thus have some of the advantages of both at the cost of having to keep the two representations consistent: a textual representation is chosen to describe scenarios in a first phase where the user is involved heavily. Then the narrative scenarios are formalized in a graphical semiformal representation. For a graphical representation, we don't want to define yet another notation and/or language. From the diagrams that are part of established development methods either activity diagrams, message sequence charts (MSC) or statecharts may be used to represent scenarios. We deliberately choose statecharts because they are hierarchically decomposable and allow for parallel states. MSCs are getting very complex and cluttered as alternatives are included. They are very well suited to represent one single path in a scenario. But they are quite problematic if a full scenario including all alternatives and exceptions is to be depicted. Activity diagrams have no clearly defined semantics. They model the control flow of a program and provide the same means for creating spaghetti control-structures as do flow charts. Furthermore, they are activity- and not event-based: as soon as an activity is done, the next action-state is entered – thus transitions are triggered by completion of actions[4]. But for many systems, more especially for reactive and embedded systems, it is a natural approach to model event-based transitions, as the systems react to external events (user inputs, sensor/actuator readings and the like). Statecharts on the other hand have a strong semantic foundation, they are apt to model system reactions to (external and internal) events and they provide the capabilities to capture a full scenario (including all its alternative flows) in one statechart. Moreover, they allow for parallelism/concurrency and for a hierarchical decomposition.

11 The Scenario Template

Having decided on a textual representation for capturing scenarios at first, we want to structure all scenarios the same way. For this reason we have established a form, a template, to describe scenarios in a consistent format[5] (see appendix A: The Scenario Template). Thus we impose a uniform structure on all scenarios, making scenarios easier to read, to compare and to use. Each single scenario description features the following fields:

- Scenario identifier

  The scenario identifier is, as the name implies, an identification label that uniquely identifies a given scenario, the identification being unique to a project. It usually will be (but does not have to be) a number or a string including numerical elements. To allow for scenario reuse across different projects, the identification number may be extended by a project- or application-specific prefix to form a unique and traceable identifier in a given domain or company.

- Scenario name

  The name is an essential part of any scenario as it implies and confers a conception, a vision, and conveys an idea of what the scenario is about, the essence of what a given scenario does, to the reader. It should be phrased as a short active verb phrase. Examples are: *Withdraw cash*, *Monitor device* or *Set room temperature*.

4. In this respect, activity diagrams can be seen as a limited version of statecharts, where transition-triggering events are restricted to 'activity completed'-events, only – call them end-of-activity events –, and states are the high-level states that represent activities. But opposite to statecharts, the semantics of activity diagrams is not well-defined. However, the use of activity diagrams to depict scenarios is enticing as scenarios are activities, but because of the mentioned disadvantages of activity diagrams we dissuade from using them for this purpose.
5. The template is based on work done in the Working Group on "Scenario-Based Requirements Engineering" of the German Informatics Society published in a technical report [Arn98], on scenario templates and description schemes come across during the interviews and reviews held for this report, and on the "Basic Use Case Template" by A. Cockburn [Coc96].

- Scenario description, purpose, goal

  In this short paragraph the intent of the scenario is summarized. What is the goal of the scenario, what resulting value does the triggering actor get? Use short active verb phrases with a clearly defined subject.

- Actors

  Who is interacting with the system in this scenario, who is triggering the scenario, whom does the scenario deliver results to?

- Preconditions

  What conditions have to be met before the scenario is executed? What presuppositions have to be guaranteed that the scenario can be run or that the scenario will deliver a correct result?

- Postconditions

  What will be the result(s) to the triggering actor and the state of the system when scenario execution stops? What will be the result(s) and state of the system if the scenario was executed successfully, what if the scenario failed? What are the results of alternative and exceptional flows?

- Trigger

  Names the triggering event that leads to scenario execution. The triggering event is usually initiated by an actor.

- Normal flow

  Description of a step-by-step procedure of the actions, events and system responses generated during normal, expected system behavior.

- Alternative flows

  Lists the flow of actions that are generated by alternative and exceptional system behavior.

- Non-functional requirements

  All the requirements that can not be captured in the (normal, alternative) flows directly are listed in this field to complete the requirements specification. Performance and timing requirements are often included in the step description, while qualities usually are documented in this special section.

  The sections of non-functional requirements in scenario descriptions have to be accompanied by a ‘global’ non-functional requirements document that captures all the qualities, constraints and non-functional requirements that apply to the whole system and not just to one scenario. That is, all requirements that can not be attributed or assigned to one single scenario have to be captured in a central document (see also the paragraph entitled ‘Non-Functional Requirements’ in section 18).

- Version/Change History

  Assigns a version number to every release of a scenario to help manage and version scenarios. Thus, different versions of scenarios may be kept. Keep a change history for every scenario.

- Owner

  The creator of the scenario, the developer who wrote the scenario and keeps the scenario up to date.

- Type

  Either “normal” or “abstract”. Abstract scenarios help to factor out common behavior, that is, common partial flows of actions that are used in several scenarios that show this behavior or use this sequence of actions are factored out in a special scenario. This sorting out of common behavior helps in modularizing scenarios, making (part-) scenarios reusable, and in keeping scenarios consistent.

In addition to the above mentioned scenario information, there are some paragraphs that are used to record information that is not directly relevant to the scenario, but that is helpful in many other respects. The information to be documented includes:

- Diagrams

  Often scenarios are accompanied by diagrams or other graphical representations like mask designs, screen layouts, screenshots and the like, that help describe the user interface, sometimes there are

pictures (of devices, environment, …) or charts (e.g. depicting the desired performance and throughput, or the linkage structure of web pages, …).

- Open Questions/Known Problems

  Often, while creating scenarios, the developer needs a place where to capture and document open issues, questions to be discussed and settled later on, pointers to things to be remembered and finally a possibility to list possible alternative paths that need to be explored in future system releases or when the normal flow of actions has been established and validated.

As a special feature of our approach, we encourage and require the developer to record and document all ideas and information that will be helpful in the future to test the system according to a given scenario. In creating scenarios and exploring the application domain, developers often 'stumble' across information and recognize important details that have to be tested later on in system test. Furthermore, as scenarios are validated, abstract test cases are developed. But in many processes there is no planned way to capture this information and preserve it from the analysis phase for the testing phase, no suitable channel is available to transmit the information from developer to tester. To avoid this problem and establish a communication path between developer and tester, we introduce a field 'Test Planning' into our scenario template:

- Test Planning, Test Cases

  To support testing, the developer documents information, that is important for testing, while eliciting and defining requirements, while creating the scenarios and while designing the system. This information may include abstract and concrete test cases, expected results, and the like. This needs not be in a structured, detailed way, it is much more important to catch all the important hints and the significant information to support the testers in achieving desired coverage. Every scenario should be accompanied by the core test cases needed to test its functionality and error handling.

The template as presented here may have to be tailored to meet specific project needs. Fields may be appended as needed. However, we suggest not to delete any of the proposed fields (except the type information field, if need be), as otherwise important information will be lost. While contents thus are fixed, more or less, the structure, layout and representation of the information certainly may be tailored to meet company standards.

12 Rules Concerning the Use of Natural Language

In the SCENT-Method we use unrestricted natural language to specify the system. However, as natural language has some major drawbacks if it is to be used as a specification language (ambiguity, impreciseness, vagueness, semantics are not well defined, can not be formally verified, hard to keep consistent or check for consistency, hard to automate, and so on), we provide some guidelines and general rules to make the user of the method aware of some of the pitfalls and to support and guide him/her in using natural language more effectively and avoiding some of the common traps. The guidance given by and the limits imposed by the rules are quite subtle and do not curtail the language inappropriately. In fact the rules are not compelling orders that have to be strictly followed, rather they are mere directives to developers; no strict rules, but guidelines, to avoid constructs that often lead to problems in specifications. As the guidelines are followed, certain constructs and words that often lead to ambiguity or misunderstandings, or that need interpretation, words that indicate omissions or impreciseness, and terms that are vague, are avoided or replaced by definite alternatives.

The guidelines are concerned mainly with single words as for example the conjunction "and" or the disjunction "or"; some describe the structure of phrases used to describe scenarios, defining a kind of well-formedness for sentences to be used in scenarios.

"AND" In natural language the conjunction "and" can carry three different meanings:

Firstly, it is used to denote the concurrency of events or actions. It can also be used to capture the fact that several conditions have to be met before an action or an event can take place (concurrence of conditions). Following, an example to each of these different cases is given: 'the phone rang and the doorbell chimed [at the same time]' - concurrent events, 'push the green and the yellow button [at the same time] to start a new run' - concurrent actions, 'to open the treasury two keys and two authentication codes are needed' - conditions that have to be met.

Secondly, a temporal order can be implied by using AND. As an example consider the following sentence: 'She opened the mailbox and took the letter. In a hurry she ripped open the envelop and read the message'. Sentences in which AND is used as a temporal conjunction can be rephrased to 'first she does something and then [she does] something else' – AND is used as a temporal link between phrases.

Thirdly, AND is used in an enumeration, implying no temporal ordering nor concurrency. Example: 'take a cup of water, a lemon and an egg' – AND is used as an explicit enumerator. The sentence may be rephrased to 'do/take/give… something, something else and yet another thing in any order you like, as is convenient to you'.

Whenever possible, phrases containing the conjunction AND should be serialized, that is, they should be transformed into a sequence of actions. Of the three mentioned different meanings of AND, the first – concurrent events, actions or conditions to be met – can not be serialized. The concurrent meaning should be emphasized by adding 'at the same time' or a similar statement. Alternatively, the logical AND-operator may be used or the word AND is written in capital letters.

Enumerations and temporal dependencies expressed by AND could be marked by adding 'in any order' or 'in this [specific] order', respectively. Preferably, they are written down as explicit sequences of statements:

"First A... and then B..." is serialized to

1. A....
2. B....

making the (often implicit) temporal ordering explicit.

Enumerations without an order, like "Take an egg and a lemon and do...", are transformed to a sequence by imposing an arbitrary order, as in the following example

1.1 Take an egg

1.2 Take a lemon

2. Do...

The steps of the sequence may be marked as an enumeration that has no implicit or explicit ordering by assigning the same number to each of the steps and differentiating just by an added sub-number (as done in the example), thus preserving the original meaning (no order implied).

"OR" Similar to AND, OR has more than one meaning in natural language. It can either mean the exclusive OR (this or that, but not both; either... or...), or the inclusive OR (this or that or both). To avoid misunderstandings and ambiguity we recommend to always use EITHER – OR for the exclusive OR and explicitly state EITHER - OR - OR BOTH for inclusive OR.

Vagueness: "several, some, few, many, …, large, small, big, fast, slow, …"

There are many more of these vague words and expressions in natural language. Avoid to use them in scenarios as they are not definite. How much is 'much', and how many are 'several'? Be specific in specifying quantities and qualities that are expected of the system. Requirements have to be quantifiable and accomplishment of goal has to be measurable.

Exception-indicators: “although, but, at times, yet, however, nevertheless, despite, whereas, …”

Words of this category indicate exceptions. Exceptions are to be captured as an exceptional flow in the alternatives section of scenarios. They should not be included in the normal flow or, if they are included in the normal flow, they have to be marked clearly (e.g. using ‘if … then … else …’ constructs and indention).

Uncertainty: “can, should, …, usually, probably, possibly, seemingly, …”

Avoid modal constructs like the ones mentioned above. If not sure of certain requirements, indicate the uncertainty clearly (e.g. by ‘to be determined’).

Avoid the use of pronouns (as they are placeholders for a noun). All to often, their reference is not clear (e.g.: ‘The baby plays with the book. It is dirty.’ The baby? The book?). Rather than using pronouns, explicitly repeat the referenced word than to leave readers uncertain about the meaning of the sentence.

Structure of phrases: Use active verb phrases with only one subject and at most one direct and one indirect object and/or a prepositional phrase, respectively. The subject has to be clearly defined and has to be an actor or the system. Do not use compositional phrases. Every step in a scenario should be a simple sentence without subordinate clauses.

Structure of scenarios: Form, layout and structure of a scenario (and partially contents also) are specified in a template thereby forcing developers to specify certain given aspects (Appendix A).

The key advice is: Be aware of different meanings of AND. Sequentialize where possible. Use EITHER OR for exclusive OR, EITHER - OR - OR BOTH for inclusive OR. Avoid modal and vague expressions. Be definite in expressing requirements. Indicate omissions explicitly (e.g. ‘to be determined’ – TBD). Use active verb phrases only.

13 Overview Diagrams

In order to convey a clear picture of the relations between actors and the scenarios/use cases and to make available a high-level, abstract view of the set of all use cases of a system, an overview diagram as introduced by [Jac92] may be created. This diagram is, however, not mandatory in SCENT and it is not further used in the method.

Actors are depicted as stickmen, scenarios as named ovals. System boundaries may be shown by a rectangle. The use and benefit of overview diagrams is quite limited, but it does not require much effort to create such a diagram. The main benefit of an overview diagram lies in the early availability of a graphical abstract view of the system’s functionality that can be used in discussing and negotiating functionality and in prioritizing scenarios. Furthermore, as a high-level abstract view of the system’s functionality that is also showing the system’s embedding in the environment, it may be used as a means to present a system overview to higher management, that is, as a graphical ‘management summary’, so to say. For an example, see appendix C, step 8 of the scenario creation procedure, or turn to [Jac92] for a more complete description.

14 Dependency Charts

Scenarios are partial descriptions of system behavior. Most often, scenarios are applicable to restricted situations, only. A scenario description of an ATM (automated teller machine) system, for example, might comprise the scenarios “Withdraw Cash” and “Inquire Balance”. Obviously, each one of these

scenarios is just capturing a small part of the system's behavior (thus being partial), and the mentioned scenarios may only be executed if the customer owns a card and knows the correct PIN (personal identification number). Thus, the scenarios are applicable in restricted situations only (only after the customer has applied for and has been granted a card and a PIN, the two mentioned scenarios may occur). As scenarios are partial models, for testing they have to either be integrated, or dependencies between scenarios have to be modeled. Otherwise the application will not be tested completely.

In most applications, the ordering of its scenarios is at least partially not arbitrary. Some scenarios must occur before some others may take place (for example initializations have to be performed before the system is to be used). Furthermore, scenarios are related to and dependent on other scenarios (e.g. money may not be withdrawn except the dispenser has been filled or refilled first).

There are copious dependencies among scenarios: Some scenarios must be executed only if certain conditions hold, like for example, that another scenario is executed first. Other scenarios may occur a number of times in a row, while still others may be executed only once. Again, other scenarios may be executed alternatively, depending on some condition. Finally ample timing conditions between scenarios may exist, as, for example, some scenarios may be executed concurrently and must be synchronized, other scenarios must not be executed in parallel. Dependencies between the different scenarios of an application exist and need to be modeled.

The need to capture the dependencies between scenarios arises more prominent, if scenarios are to be used to develop system tests. If scenarios are used as a means to improved communication with the customer, or as a helpful means to elicit requirements, the dependencies might not be that crucial and scenarios may be used without a notion and notation to capture dependencies among them. But in testing, dependencies are most crucial, they stand central to complete testing. This means that no test can yield satisfactory results if dependencies between scenarios are not considered and tested for.

Furthermore, a hierarchical decomposition structure for scenarios is desirable and even required in developing huge systems. Business cases, being a sort of scenarios themselves, are composed from task-level scenarios. These in turn may be composed from simple building blocks that are (partial) scenarios themselves (thus allowing for reuse of scenarios as well). To depict the compositional structure of scenarios, a model that shows hierarchical structure, interrelations and dependencies between scenarios is needed.

Dependency Classification

Dependencies among scenarios may be of different kind. We distinguish three major kinds of dependencies, all of which are related:

- Abstraction dependencies. Dependencies of this kind are introduced into the model in different ways, however, all of them represent some form of abstraction. Examples are dependencies introduced by hierarchical decomposition of model elements, by aggregation, generalization and refinement. Scenarios arranged in hierarchies, scenarios to cover variants (e.g. the same scenario with various slight differences is true for a system depending on hardware configuration), scenarios composed of sub-scenarios and the like, all of them establish abstraction dependencies.
- Temporal dependencies. A temporal dependency between scenarios exists if scenario A has to be executed before/after/at the same time as/at a given time/at a given time relative to the start or the end of or in parallel to scenario B. Temporal dependencies establish a sequence dependency between scenarios. They map to strict sequences or to real-time dependencies in dependency charts: Scenario A must be followed by scenario B (see description of strict sequences below). In Figure 4, an overview of possible time dependencies is given. The length of the boxes in Figure 4 indicates relative scenario duration, the x-axis represents time. Starting and end points may be absolute or relative.
- Causal dependencies. If scenario B may only be executed under certain conditions and scenario A establishes these conditions, then the two are related by a causal dependency. Causal dependencies

usually establish a loose sequence in dependency charts, that is, scenario B may be executed any time the conditions hold true. Once scenario A has been executed, scenario B may be executed. Causal dependencies are very similar to temporal dependencies, in fact, they always can be mapped to temporal dependencies. However, temporal dependency are more strict than causal dependencies: while a scenario related by a temporal dependency will be followed by the dependent scenario, in a causal dependency, one scenario but establishes conditions that the dependent scenario may be executed. Thus, a temporal dependency exists between a calling and a called scenario, a causal one exists if scenario A by flipping the condition that triggers scenario B, causes B to execute. All dependencies that relate to data having to be created or prepared first, or certain value to be reached, all flags, state dependent behavior and execution, decisions depending on testing a variable or value set in another scenario and the like, are subsumed under the term *causal dependencies*. For example in an ATM system, the user having logged in to the system first, before performing any ATM-function, establishes a causal dependency between the 'Authentication'-scenario and the 'Withdraw cash' and the 'Inquire balance'-scenario, respectively.

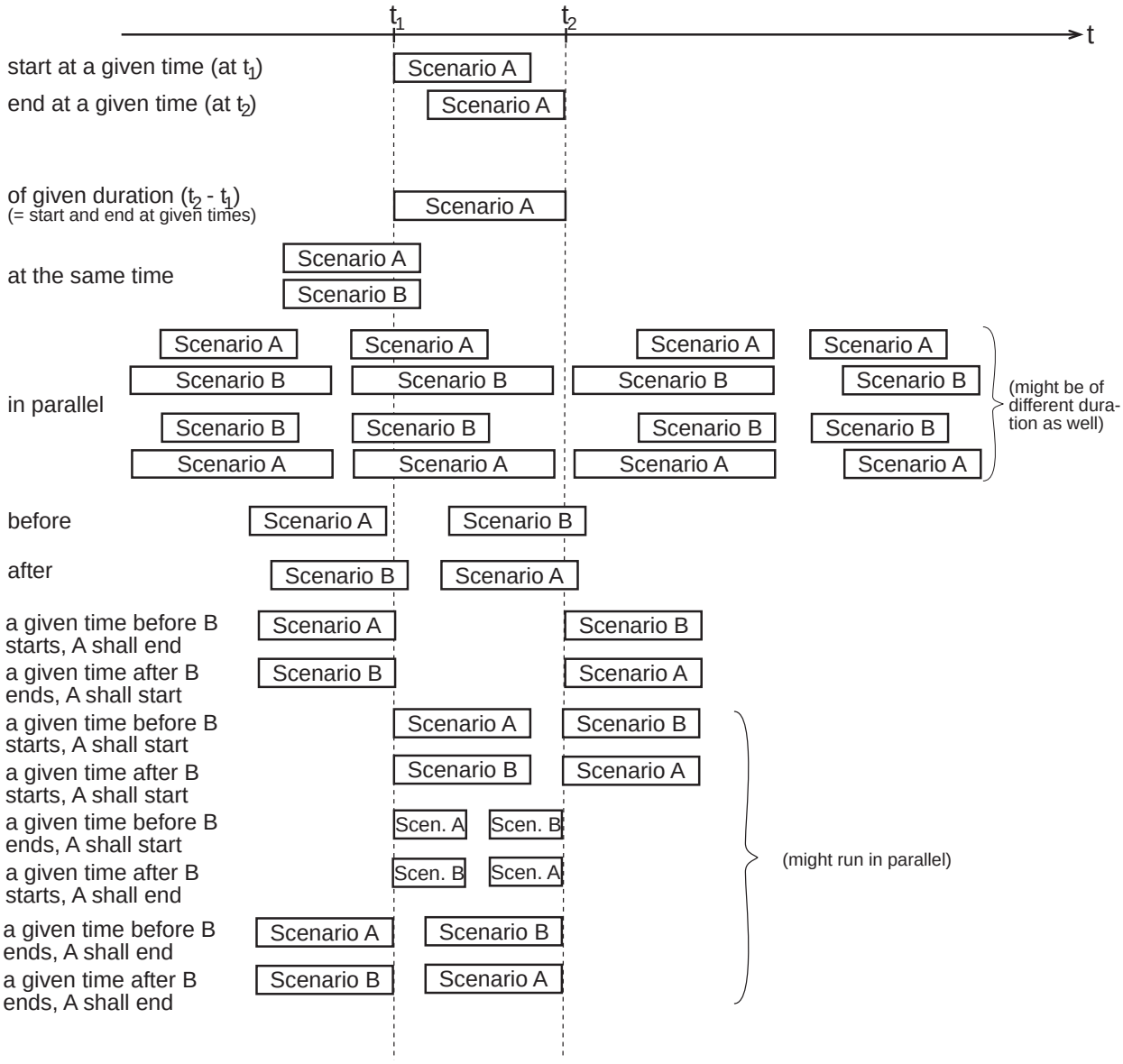


FIGURE 4. Temporal dependencies of and among scenarios

Most temporal and causal dependencies may be captured by execution order. With respect to their execution order, scenarios may be related one to another in four ways: one scenario follows the other one (sequence), either one or the other scenario is executed (alternative), one scenario is executed multiple times (iteration) and a scenario runs concurrent with another one (concurrency). Data and resource dependencies need to indicate what data items and resources are concerned. Abstraction relations may be shown in an abstraction hierarchy.

The categorization given above should be used as a help to developers to remember where to look for potential dependencies and to ask appropriate questions (e.g. “does this scenario have to be preceded by another scenario, does it always follow another scenario?”, “What conditions have to be fulfilled, that the scenario under consideration is executed? Which scenarios establish these conditions”, “What data does the scenario work on? Where does the data come from, who is preparing it?”, a similar question may be asked for resources, …).

Notation

To depict dependencies between scenarios, an existing notation, e.g. the scenario overview diagram of Jacobson et al. [Jac92], might have been chosen and extended appropriately[6]. The original concepts and notation for overview diagrams does not allow to capture structures like sequences, alternatives and iteration. Thus, it would have been necessary to extend the language. This, however, might lead to confusion and misunderstandings. For this reason, it is advisable to use a distinct notation to emphasize the difference of the new diagram from existing ones in meaning and use[7].

In order to capture dependencies among scenarios we introduce a new type of diagram which we call *Dependency Chart*. On the one hand, dependency charts are used to help the developer gain a clear understanding of the system’s high-level dependencies and connections between scenarios, thus supporting the development of more accurate and meaningful models of the system. On the other hand, dependency charts are used to derive additional test cases to enhance the test suites generated from the state model of the system.

We want to depict dependencies between scenarios in a graph: the nodes shall represent the scenarios, the edges shall represent the dependencies. Sequence, alternative, iteration and parallelism shall be represented in an expressive way. Dependencies of the different types – temporal, causal and abstraction – shall be represented and the notation shall be intuitive.

In dependency charts, scenarios are shown as round-cornered rectangles[8] with connection circles attached to both small sides of the rectangle. The connector circle represent the entry and the exit point(s) respectively (Figure 5). We call these structures *scenariolines* to have a distinct term for the

6. We point to the fact, that overview diagrams were not intended to be used to capture and depict dependencies between scenarios, and that the notation as proposed by Jacobson and used in the UML is not sufficient to do so, but would have to be extended. The only relations that are depicted in overview diagrams are:

   - associations between actors and scenarios (an actor participates in a use case),

   - use/include relationships (the behavior of the included use case is part of the including use case as well),

   - extend (use case B is extended – under certain conditions specified in the extension – by the behavior of use case A, thus the extend relation represents a conditional include), and

   - generalization/specialization relationships between use cases.

   General dependencies, time and data dependencies and sequencing, including alternatives and iterations, are not depicted in overview diagrams. However, the notation could be extended to include the necessary structures to depict general dependencies, but the original intension of the diagram would be lost in doing so.

7. Further discussion of dependency charts and alternative notations may be found in this section in the paragraph entitled “Why define a new notation?” on page 42.

8. The rectangles have rounded corners to show resemblance to the ovals of scenarios in overview diagrams, yet to have distinct characteristics

graphical representation of scenarios in dependency charts. Scenariolines may be drawn horizontally (normal case) or vertically, but all scenariolines are drawn in the same direction. Scenariolines that are not connected to any other scenariolines by dashed or regular lines (dependency lines) are unrestricted. We call them 'unbound'. Unbound scenarios can be executed as many times as desired and in free order, even in parallel with other scenarios, if appropriate. The two unbound scenarios in Figure 6 may be run in parallel, they may follow one another in any order, or one or the other, or even both, may not be executed at all. They are not restricted to a specific sequence, neither do time, data, resource or other dependencies to other scenarios exist.

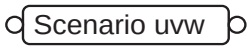


FIGURE 5. A scenario in a dependency chart (a scenarioline)

Take note that the notation does not imply that one of the scenarios is executed first, but assumes that either scenario may be run first or both may be run in parallel, and that it does not imply that the scenarios have to be run at all: The scenarios may, but they don't have to be executed.

The (horizontal) position of scenariolines indicates the relation of the scenarios one to another. If scenarios may be executed without any restrictions on sequence, that is, if they are not required to be in a specific order, then scenariolines are drawn in parallel, all the rectangles aligned to the left (Figure 6). If scenarios are to be executed in a specific order (sequence), they have to be arranged accordingly (see the paragraphs below and Figure 7). If the execution order is not definite, but a certain order is more likely than another, this fact may be indicated by indenting the scenariolines (Figure 15 and accompanying text). The length of the scenariolines does not carry any meaning, in particular it's not indicating duration of execution of the scenarios. If entry or exit points are connected by dependency lines that run perpendicular to the scenariolines it indicates simultaneousness, that is, the two scenarios start or end at the same time (Figure 14). Vertical spacing and positioning of scenariolines does not carry any meaning.

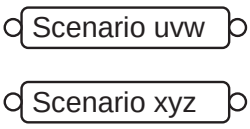


FIGURE 6. Independent, 'unbound' scenarios

Sequence, Alternative and Iteration

A sequence of scenarios is shown in dependency charts by a sequence of scenariolines (Figure 7). Scenariolines appended one to another thus express that the first scenario has to be finished before the second scenario is executed. This is often the case, especially in initialization sequences, as one scenario often collects, computes and prepares the data needed by a second scenario.

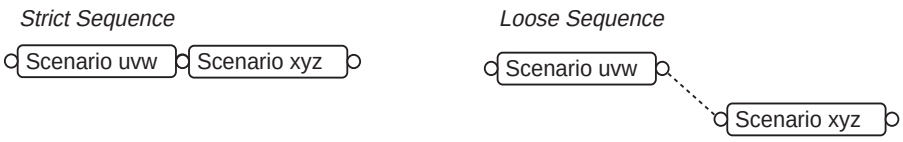


FIGURE 7. Scenario sequences in dependency charts

We distinguish between strict and loose sequences: in strict sequences – indicated by directly connecting the exit-point of one scenarioline to the entry-point of another scenarioline – the second scenario has to be preceded by the first scenario and the first has to be followed by the second, in loose sequences – shown by a slanted dependency-line (dashed line) – the first scenario has to be executed before the second is run (that is, if the second is to be executed, the first has to precede it, but the first may well be per-

formed without the second). An example of the former is the scenario of applying for a card for access to a system (e.g. a library card, a bank card to use at an ATM, and so on) which has to be followed by an *issue card* scenario in the normal flow of actions. An example of the later is the scenario of doing statistics on data collected in another scenario: the 'doing statistics'-scenario depends on the data collecting scenario to be executed first, the data collection scenario however is fully independent from the statistics scenario (it does not have to be followed by the statistics scenario).

Alternatives are shown by a scenarioline splitting into two (or more) scenariolines (fork). The fork is graphically shown by a connecting line from the scenario exit point of the scenario preceding the alternative to the entry points of the scenarios that are executed in dependence of the alternative taken. The line is drawn rectangular to the scenario baseline. Forking is not restricted to binary alternatives (Figure 8). Alternatives may be annotated with the condition(s) and alternatives that can be taken. This is not mandatory, however. Often alternatives are quite obvious in naming of the alternate scenarios. Thus, conditions are only specified in dependency charts if needed. As scenarios may be triggered and executed on users choice (in reactive systems), all scenarios are alternatives depending on the user's choice. This fact is not modeled (see for example Figure 18: even though the scenarios *Withdraw cash* and *Inquire balance* may be executed alternatively, they are just shown as unbound scenarios. Unrestricted scenarios by definition may be executed alternatively, even in parallel). Only alternatives in a sequence that must not be taken out of context and can't be decided on independently are modeled.

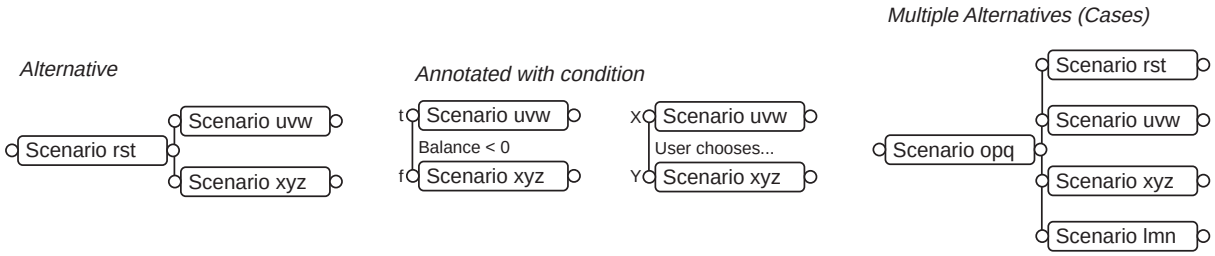


FIGURE 8. Alternatives in dependency charts

Dependency charts also offer a way to model iterations. Iterations are shown by backward sloping arrows connecting the exit-point of one scenarioline with an entry-point of any (not necessarily the same) scenarioline in the same sequence of scenarios. An iteration may encompass as many scenarios as desired as long as the scenarios are all in the same sequence. An iteration condition can be attached to the arrow-line: Using the well known 0, 1, * notation to denote multiplicity, an absolute number of iterations may be specified, other iteration conditions can be expressed using logical expressions and arithmetic (e.g. 'while balance > 0', or 'until number of participant >12' for the hypothetical scenario "Register Participant"). However, iterations often are implicit: Any unbound scenario or scenario sequence in a dependency chart may by definition be executed as many times as desired or needed.

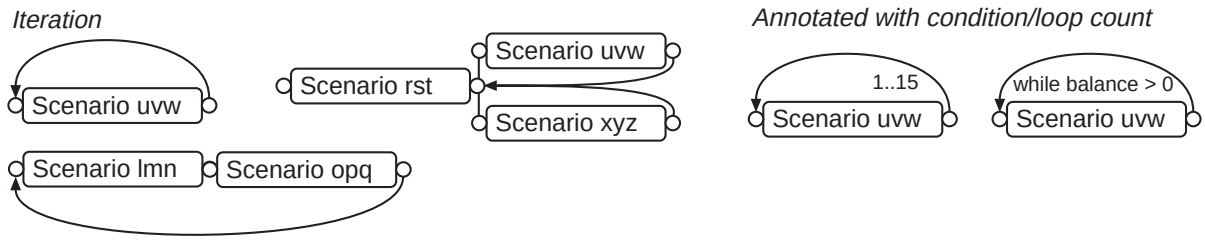


FIGURE 9. Iterations in dependency charts

Further Structures: Time and General Dependencies, Structuring Constructs

Abstract scenarios (as factored out in step 12 of the scenario creation procedure) do not have to be shown in dependency charts. They may be shown, however, as foldouts, marked as abstract by dashed borderlines (Figure 10). Abstract scenarios are to be self-contained in order to be plugable.

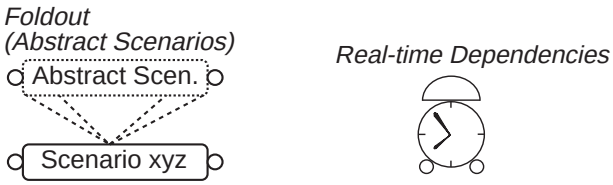


FIGURE 10. Abstract scenario represented by a foldout, real-time dependencies

Real-time dependencies are indicated by the alarm-clock symbol (Figure 10).

General dependencies among scenarios are shown by dashed lines, that is, annotated dependency lines are used to depict any other dependency (general dependencies, data/resource dependencies). Dependency lines should be named or annotated with needed information where appropriate. Some examples are provided in Figure 11. Dependency lines may be directed to indicate the dependent scenario. In this case a dashed arrow line is used to represent the dependency. The scenario at the tail of the arrow depends on the scenario at the arrowhead. In Figure 11 in the example to the right, scenario uvw depends on scenario xyz (e.g. scenario xyz calculates data that is used by scenario uvw).

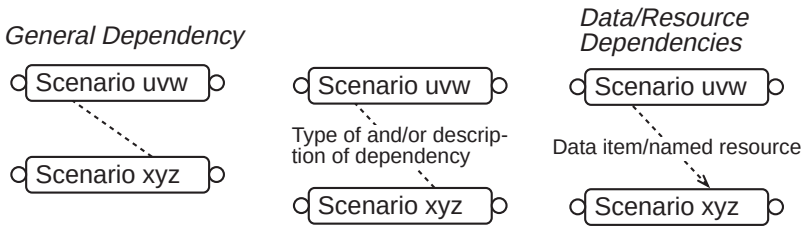


FIGURE 11. General dependencies

To structure the dependency model we use a hierarchical structure. Scenarios which belong together according to some criterion may be packaged in a box (Figure 12). This structural mechanism is very important as large models need to be decomposed into smaller pieces. The chunks that belong together are identified by the dependencies between them. As is the case in modularizing a program, the goal is to keep coupling between packages low and to have cohesive packages.

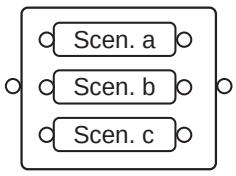


FIGURE 12. Structuring construct in dependency charts

Business processes define a first set of packages. On the next level down, workflow and dependency structure between scenarios often define the packages in a natural way. If further decomposition is needed, the concurrent and repeating blocks in the scenario structure lend themselves for a logic division of the model into hierarchically decomposed parts of the model (for an example see Figure 17, where decomposition blocks are determined by repeating blocks: in the scenario flow, the scenarios that

are concerned with the registration and deletion of users and books enclose the scenarios of lending and returning books. The repeating blocks are marked by iteration arrows on the decomposition blocks). In structuring large models using the packaging structures described above, the developer has to be careful not to make a functional decomposition. Scenarios are a functional view on the system, yet they represent a logic unity, a flow of actions from triggering event to delivered result and thus, they may comprise or make use of many program functions. Decomposition in dependency charts should strictly be limited to help structure the model by dividing the model into pieces that easily can be perceived, understood and handled.

Concurrency, Probable Execution Order, Execution Frequency and Alternative Flows

Finally, concurrency has to be show in dependency charts. As unrestricted, unbound scenarios may run in parallel, accidental concurrency is not explicitly shown. If concurrency has to be enforced or prevented, the scenarios are connected by "have to be executed in parallel" (=) and "no parallelism" (≠) marks, respectively. This means that unbound scenarios (which is the normal case) may, but don't have to, be executed in parallel (concurrency being an accidental feature), scenarios marked with ≠ must not be run in parallel and scenarios marked with = must be executed in parallel (Figure 13). We point to the fact that the use of the terms concurrently/concurrency and parallel/parallelism in this context carries a restricted meaning: they denote only high level simultaneousness which down to execution on a real machine might well be broken down to sequential working steps from different scenarios that are interlaced, thus giving the impression of parallel, simultaneous execution by mere rate of speed.

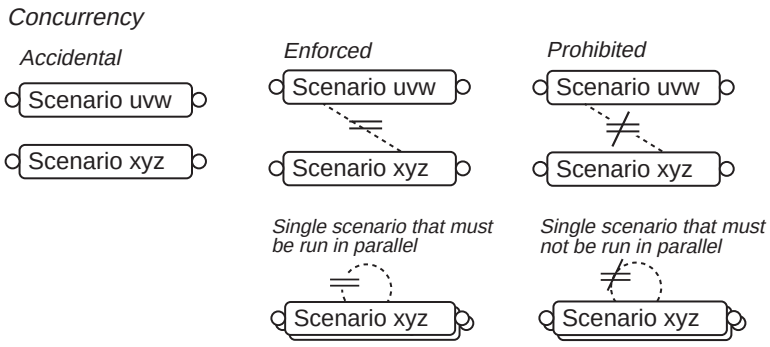


FIGURE 13. Concurrent scenarios

To keep diagrams uncluttered "no parallelism" marks may be shown not as connectors, but as annotations listing the scenarios that must not be executed in parallel. Furthermore, it is not mandatory to mark single scenarios that must not be run concurrently with an "no parallelism" mark.

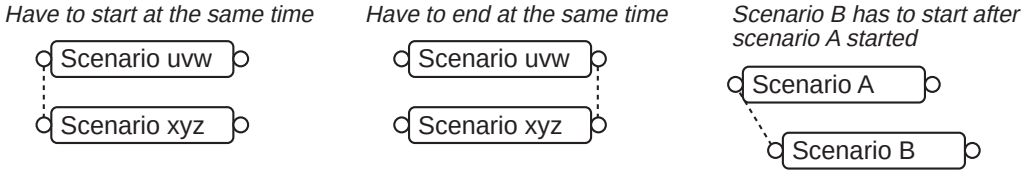


FIGURE 14. Special cases of concurrency

Many different shades of concurrency of scenarios can – if necessary – be made explicit in a dependency chart: Scenarios that have to start at the same time are connected by dashed dependency-lines connecting the entry nodes of the two scenariolines, the connecting line being rectangular to the baselines of the scenariolines (this is equally true for more than two scenarios). Just in the same manner, scenarios that have to stop at the same time are marked by a dashed dependency line from the exit-point of one scenario to the exit-point of the other scenario (Figure 14). If scenario A always has to start before

scenario B and the two will run concurrently, they are connected by a slanted dependency line from the entry-point of scenario A to the entry-point of scenario B.

Indent of scenariolines indicates that the indented scenario normally will be executed after the unindented scenario. Again this is a may, not a must. To depict the fact that a given scenario has to be run only after another scenario has been executed, use sequences (see above). The position of scenariolines thus indicates expected, but not mandatory execution order (Figure 15). Longer distances do not connote longer times in-between the execution of the scenarios. The (horizontal) spacing, though on the time axis, does not carry the meaning of absolute or measurable, comparable relative times (but instead the time axis may be seen as carrying an ordinal scale). That is, if scenario B is indented twice as much as scenario A, and A is executed after t seconds, B does not have to wait for twice the time to elapse to be executed, that is after 2t seconds only. Indenting is an optional feature in dependency charts. It carries information that helps model a usage profile for the application. Currently, however, this information is not used in deriving test cases from dependency charts.

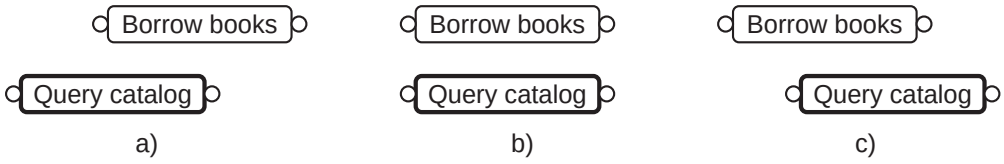


FIGURE 15. Indenting scenarios to indicate likelihood of execution order

An example of indenting scenariolines and the implication thereof: In a library the user usually will query the catalog first, to obtain the catalog number of the book he is looking for, and then borrow the book (Figure 15a). The *query* scenario will be executed more often and usually before the *borrow books* scenario. The probable execution order is indicated by indenting the scenario that normally (but not mandatory) follows the other scenario. As has been said, this sequence is by no means mandatory: The user may directly borrow a book without querying the catalog (e.g if she spotted the book she looked for on the shelf already, or if he borrows a book chosen be browsing through the bookshelf and picking interesting books going by promising titles). Or she may first borrow a book and then search the catalog for related titles. Some user might just search the catalog and not borrow any book, probably because the book sought for is not available. Thus all sequences and combinations are possible, the indention merely indicates expected or normal (most frequent) sequencing.

On the premises made, the appropriate way to model the dependencies between the two scenarios is as shown in Figure 15a. But if the *query* scenario is not more likely executed first, then Figure 15b is the right way to model the scenarios in the dependency chart. In Figure 15c the assumptions are inverse, normally a book is borrowed before (if at all) the catalog is queried. Note that for indenting only the likelihood of a specific ordering (in time) is considered and not the relative frequency with which a certain scenario is executed.

Frequency may be indicated by the heaviness of the border of scenariolines. The heavier a border, the more frequent a scenario is executed. Frequency may be estimated based on expectations or based on experience (in similar applications, in a former release of the same application and so on) or on survey results. By this precept the more important scenarios (as to their frequency) are graphically highlighted (Figure 15).

If importance and/or priorities of scenarios that are not reflected in frequency are to be shown in dependency charts, shadings and background patterns can be used in scenariolines. What has been said of indenting, is true for shading and use of patterns: the information captured this way currently is not used in deriving test cases.A special kind of dependencies is created by alternative flows and error handling as often special scenarios are called from alternative flows. While these scenarios may be shown as alternatives in dependency charts, it proves helpful to have a way to distinguish normal and alternative flows in dependency charts as well. Therefore, we suggest to mark alternative flows and error handling

procedures by decorating the dependency line with a distinct adornment at both ends of the line. Furthermore, alternative and exceptional flows may optionally be emphasized by attaching a specific pictogram to the dependency line (Figure 16). This is especially appropriate if no condition or description of the alternative flow is given.

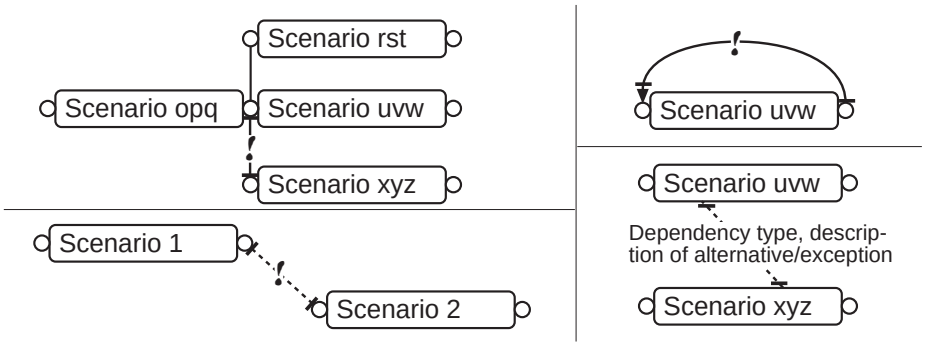


FIGURE 16. Dependency lines indicating alternative flows

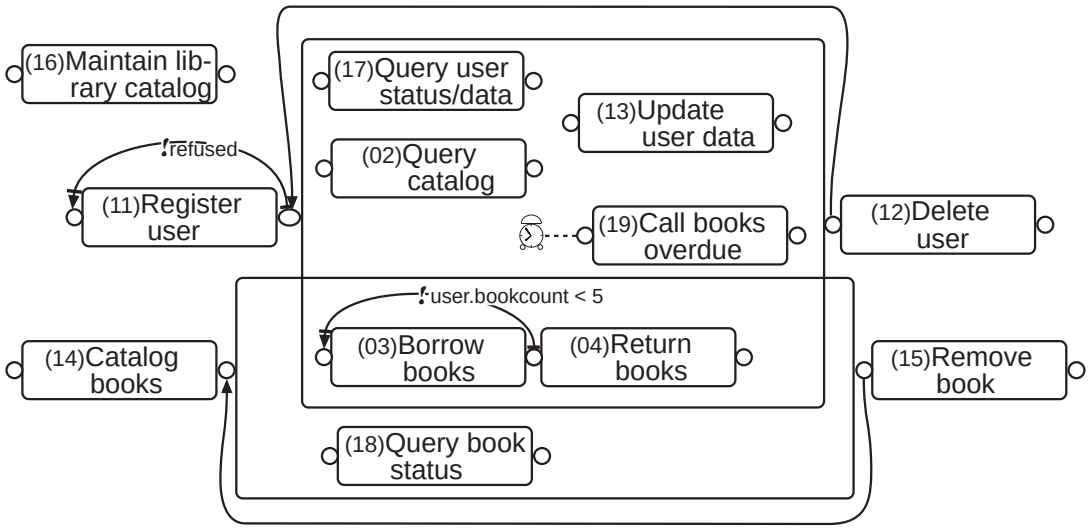


FIGURE 17. An example dependency chart

Control Scenarios

Scenarios that capture the structure and collaboration of other scenarios instead of describing the steps of the interaction between system and actor(s), we call ‘control scenarios’. They capture only control flow information and calls to other scenarios, and do not model independent sequences of working steps. In the ATM example a control scenario might capture the fact that first a card has to be applied for (scenario ‘Apply for card’), then the card is issued (‘issue card’) (or the application for a card is turned down) and only after the card is issued, the scenarios “Withdraw Cash” and “Inquire Balance” may be executed[9]. Consequently the control scenario “Get and use bank card” would read:

9. While it is true in the example, that in real life the restriction is enforced by having or not having a card, this will not be normally the case: restrictions are not enforced by tangible artifacts usually...

1. Apply for card
2. Issue card
3. Withdraw cash **or** Inquire balance
4. **Return** to 3.
5. Card destroyed or expired

In control scenarios, a sequence of scenarios is shown by the sequence of steps (‘1. Apply for card’ before ‘2. Issue card’ and ‘3. Withdraw cash’), alternatives are shown by the keyword **or** and iterations are triggered by **return** statements. If alternatives have to be specified in more detail (e.g. describing the conditions as well), an **if** ... **then** ... **else**-structure may be used.

What is described in a control scenario, is just part of the information depicted in a dependency chart (Figure 18). Control scenarios are easily mapped to structures in dependency charts. However, a graphical representation of the dependencies often is more concise and comprehensible. And more than that, in dependency charts all kinds of dependencies are depicted, not only the control flow, as is done in control scenarios.

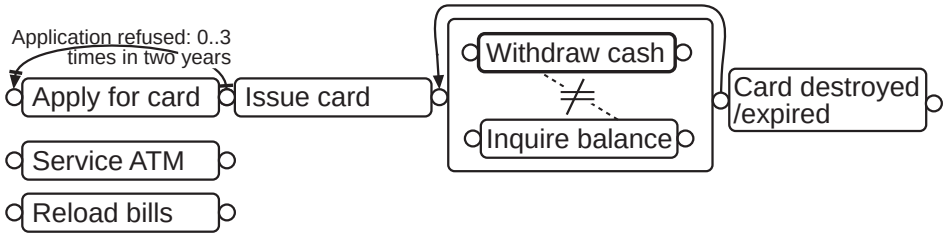


FIGURE 18. Control scenario in the ATM example

Intra-Scenario Dependencies

In this chapter, we have talked about inter-scenario relations and dependencies, that is, dependencies and relations between two or more (different) scenarios. There are also intra-scenario dependencies – meaning dependencies that just affect one scenario. An example of the later is that data first has to be fetched and processed before it can be displayed, or that a wrong password may only be entered three times in a row before the log-in procedure is canceled, the account is locked and the system administrator is informed. These facts do not establish dependencies between different scenarios, but they determine dependencies in one scenario (e.g. that the step of fetching data has to precede the step of displaying data, or that a count on the unsuccessful log-in attempts of a user in a given period has to be remembered). While these internal dependencies are important, they are not captured in dependency charts, but rather they are captured by specifying the sequence of working steps in a scenario and by alternative and exceptional flows.

Obviously, what is considered to be on intra- and what on inter-scenario level depends on the abstraction level and the application domain of the observer/viewer. While for most applications it will not be necessary to specify detailed data access on a hard disk drive, for the developer of an operating system or of a hard disk driver this will be crucial (and thus it will be necessary to have scenarios that distinguish fetching data from cache or from disk, specifying the dependencies that exist between these two scenarios, while these scenarios and dependencies normally are subsumed in one single working step in a scenario in most other applications).

Scenarios are sequences of actions or interactions that are closed in themselves; that means that a scenario is a sequence of actions from start to end and has to be self-contained and complete in the sense that a desired result or a specific use/benefit/advantage for the actor has been achieved. However, granularity and scope of scenarios depend on the purpose and use of the scenarios, and on the level of abstraction. What is considered to be atomic in one setting (for example from a user’s point of view) will be but

an intermediate state in another setting (for example from the viewpoint of a developer or of a tester). Scenarios that are considered atomic for one task often will (have to) be further refined and decomposed for another task. Yet, at the level of abstraction that is of interest to a given stakeholder in the specification and development process, and from the viewpoint of what scenarios are intended to be used for, scenarios are perceived as logical entities that can not be further divided without loosing the full picture. It does not make sense to tear apart the steps in a scenario from an operational view (this does not preclude that it may make sense to factor out abstract scenarios from a maintenance point of view or to keep scenarios consistent).

As much as it does not make sense to further divide scenarios (from a user view), it is not sensible to further integrate scenarios to represent the full system. The advantages and the goals aimed for by using scenarios are confounded in integrating them. User focus is lost, the focus on a single scenario of the system is blurred and comprehensibility is inhibited.

Scenarios could easily be further divided into (recurring) scenario fragments. In fact scenario patterns could be defined and used to build scenarios from predefined fragments.

Why define a new notation?

In software development there exist many methods that use graphical model representations. Why not use an existing notation to depict dependencies between scenarios?
In the following section we contrast dependency charts with some of the well-known modeling languages, highlighting the similarities and differences between them and pinpointing some of the problems if the existing languages were used.

In describing scenarios and dependencies among scenarios we use the well-known constructs and structures of structured programming, namely sequence, alternative, iteration and parallelism. Thus all the notations used to model control, data or work flow, more especially all diagrams used to model structured program flow, could be used as a basis notation for capturing scenario dependencies. In fact, as has been proposed in [Gli95], a Jackson-like notation can easily be adapted and used to model scenario structures, compositions and sequences. The Jackson notation needs but to be extended by a construct to model parallelism (introduced in [Gli95]) and by a concept to model block-calls / subroutine-calls. Block-calls are needed to model calls from within one scenario to another scenario. The integration of abstract scenarios is achieved this way also. Blocks are identified either by the same name or by use of numerical unique identifiers: Any block having the same name or the same identifier as another block is referring to the same model element. Both of the mentioned extensions are easily introduced in Jackson's notation.

Why then do we not just use Jackson's notation to also depict dependencies between scenarios (inter-scenario dependencies)? There are several reasons. First of all, scenarios are not easily put in a sequence, an execution order. They often are highly parallel, a given execution order is only probable, in fact even that a scenario is executed is but assumed, but not certain. Jackson's notation is restricted to model sequences, alternatives, iterations, and, with an appropriate extension, concurrency. It does not, however, allow for the modeling of further dependencies, like for example the fact that books can be added and library users can be registered independent one from the other, yet a book can only be borrowed by a given borrower if the book has been cataloged **and** the borrower has been registered. In Jackson diagrams it is hard to model different view-points in an integrated model. For example different user privileges or user groups, behavior that is state-dependent, the normal and alternative flows in a scenario, or scenario and procedure calls, respectively, are not easily captured in a Jackson-diagram. Further problems in a Jackson-like diagram arise if scenarios are not single-entry, single exit. The integration of alternative flows, creating further dependencies, in the same diagram as the normal flow and not loosing clearness and comprehensibility is hard to achieve in a Jackson-like notation. A 'flow notation' is much more suited for this kind of task as it accommodates depicting alternatives in a way that viewers can quite easily follow them. However, the cost of using a flow-chart-like notation is the loss in

structure: any connection in-between scenarios is possible, even if the depicted flow in reality does not make any sense. The layout of elements is not that clear anymore, and the graphs may be quite cluttered.

Depicting concurrency in Jackson-diagrams poses many problems as well. Does parallelism for scenarios mean that the scenarios start at the same time and have the same duration, or does it mean that they have to start at the same time only? Does it mean that they have but an overlap in time of execution, that they need to have certain competing steps, or that one scenario needs to start before and end after the other one? Thus, to depict dependencies between scenarios, a differentiating notation that helps depict all these shades of parallelism is desirable. Furthermore, only the leafs of the tree-structure are scenarios (or compose a scenario), the inner nodes are but control structures (composing kind of a control scenario). Thus unnecessary complexity is added as in dependency charts we are concerned about and interested in dependencies and relations between scenarios only.

As in scenarios the exceptional flows and other dependencies (data, time,...) are just as important as the sequences, alternatives or iterations of the normal flow, we rather use a new notation that is tailored to these demands, namely dependency charts, instead of using Jackson-diagrams that have to be extended and adapted to the requirements of depicting dependencies between scenarios.

Why not just integrate dependency charts and the overview diagram? Because they serve very different purposes. The overview diagram is to give a short graphical summary of the scenarios of an application. If dependencies were to be included into the overview diagram, it would get cluttered and unfit for the purpose it serves. Dependency charts on the other hand are less apt to be used as an overview chart: they include too much detail, show relations between scenarios, but do not show the setting and embedding of the application in context (actors, external systems, sensors, actuators). For this reason, we argue that a special type of diagram is needed to depict the dependencies among scenarios.

In many respects dependency charts are similar to Petri nets: They depict the non-determinism, parallelism/concurrency and the synchronization between scenarios. But dependency charts depict mainly the statics and less the dynamics of the relations between scenarios. Thus they are easier to understand than Petri nets. Furthermore, dependency charts allow for the notion of probability: If scenario A is likely to be executed before scenario B, this may be indicated by indenting scenariolines. And what's more, if scenario A is expected to be executed more often than scenario B, this can be indicated in dependency charts as well. Dependency charts are not bipartite graphs as Petri nets are. Dependency charts know only one kind of knots, namely scenariolines.

Dependency charts also represent a kind of flow chart, presenting a distinct facet of control flow, namely the structure of control among scenarios. Dependency charts do however include information that is not specified in flowcharts: Time dependencies, parallelism, data dependencies and so on. Furthermore, the differences mentioned above (Petri nets) hold true for dependency charts as compared to flowcharts as well. To graphically distinguish dependency charts from other forms of flow charts a distinct notation has been chosen, that helps to clearly arrange the elements (scenariolines) and to show the dependencies among them.

Dependency charts are especially apt to picture temporal dependencies. As such they are closely related to and have quite some similarities with constructs and graphical notations for temporal logic. We do not however enforce a strict formal notation, and some of the underlying semantics in dependency charts is quite weak, as dependency charts just serve as a graphical overview of all existing dependencies among scenarios. The details of dependencies are specified in any desired language using formal notation as desired and appropriate in the scenario description.

As an alternative to dependency charts, dependencies between scenarios may be captured and described in natural language, for example in a tabular form. The disadvantages of doing so have been discussed in some detail in section 10 where graphical and textual representations are contrasted. However, to give an impression of what it would look like if a textual representation were used, we specify just a couple

of dependencies in a tabular form. The scenario numbers relate to the example given in Figure 17, scenario 20 is the (supposed) scenario of “Reserving a borrowed book”.

TABLE 4. Natural language representation of dependencies

Dependent Scenarios	Dependency Description in Terms of Scenarios involved	Informal Dependency Description	Formal Dependency Description
...	...	...	...
04, 12	Scenario 12 may not be executed, before scenario 04 ended successfully	Books borrowed by user A have to be returned before user A is deleted	¬User_A.Bookcount =0 -> ¬Delete(User_A) ∀ User_A ∈ Users
03, 04	Scenario 04 may not be executed before scenario 03	A book can not be returned before it has been borrowed	¬Book_B.OnLoan = true -> ¬Book_B.Return ∀ Book_B ∈ Books
03	Scenario 03 may not be executed if User_A.Bookcount>5	Any user may borrow five books at most at any time	User_A.BookCount <= 5 ∀ User_A ∈ Users
03, 20	Scenario 20 must be preceded by scenario 03	No reservations on a book that is not on loan	Book_B. ReservationCount = 0 ∀ Book_B ∈ Books.NotOnLoan
20	Scenario 20 may not be executed more than three times for any book on loan	No more than three reservations on a borrowed book	Book_B. ReservationCount <=3 ∀ Book_B ∈ Books.OnLoan
...	...	...	...

It is quite obvious that dependency charts have to be supported by a tool to use them effectively. The multitude of concepts and plenitude of graphical elements and attributes carrying each one their specific meaning makes the proper use of dependency charts a challenging task if done without tool support. As a proprietary notation is chosen, existing tools can not easily (that is without tailoring) be used to draw dependency charts. Yet there are ample benefits in using a dedicated chart to depict the dependencies between scenarios. Dependency charts are of great importance to use the full power of scenarios throughout the development cycle as dependencies between the different uses of a system are of major interest in design and of uttermost significance in testing and maintenance.

CHAPTER 6 *Scenario Validation, Transformation and Annotation*

In the SCENT-Method, requirements are captured in narrative, natural language scenarios at first. Requirement specifications that rely on natural language may be incomplete, ambiguous and/or inconsistent. They may rely on implicit knowledge, meanings, interpretations and assumptions made by the specifier, "obvious" to himself, but not to the designer or tester. Specifications often rely on common background (knowledge) and common understanding. Specifications and requirements can be misunderstood and interpreted. And finally requirements are changing, thus invalidating specification and design. Consequently the design and the code has to be changed. All these factors contribute to bugs in a system [Bei90].

It is therefore imperative and indispensable to validate the specification to make sure that it adequately and completely states the needs of the users. Moreover, it is equally important to check that the specification is consistent and complete and that it is not ambiguous or vague. Understandability and testability are further key properties that a specification needs to be checked for.

Furthermore, not only the specification (including the scenarios), but also all the documents and models derived from scenarios (as for example class and behavioral models in design) need to be verified and validated. While validation can hardly be automated, verification of the system may be supported and even partially be automated by the models and languages chosen in design, e.g. state automata may be animated and executed, consistency constraints on SA models may be checked by a tool, and so on.

15 Scenario Validation

Natural language scenarios can, as a first validation activity, easily be "walked through" by the user. Developers should step through each scenario with the customer/user and have him/her comment on normal and alternative flows. Alternatively, scenarios may be 'acted out': Have users play the actors and the system, and step through the scenarios by triggering events and reacting appropriately. The system may even be divided into different modules and partitions. Thus, even internal behavior can be visualized. As a matter of course scenarios may also be formally inspected/reviewed. As is the case with any textual document, only static validation and testing methods can be applied to narrative scenarios.

(Narrative) scenarios can further be validated by sketching dialog masks, screen layouts and menu structures of the application to be developed. If a prototype is developed, the prototype may be validated by the scenarios and the scenarios are validated as they are worked through in the prototype.

As scenario elicitation is iterative and incremental, validation and verification activities are repeated more than once. Thus many inconsistencies, omissions, over- or under-specified requirements as well as ambiguities may be found in reviewing and by walking through the scenarios with the customer in every iteration step.

Furthermore, by formalization of narrative scenarios in statecharts (section 17), simulation and animation of the model are feasible and help to further validate the requirements, the scenarios and the model.

Scenarios are a description of the way a user works with the system. It is easier for users to validate the sequences of actions and events as specified in a scenario than it is to validate requirements presented to them in a classical specification, where requirements are listed in a classifying structure (as for example in listings of functional requirements, of performance requirements and the like). In these classical categorizing schemes, requirements are listed decoupled from the task and working steps they specify or they are specified by. They are disconnected from process and rationale indicating why they are required. Requirements arranged according to these classification schemes are often unrelated one to another, while in a scenario relations and dependencies are often intelligibly connected.

In short, use the following means to validate scenarios:

- Walk the user through the scenarios.
- Use reviews and static testing methods to validate natural language scenarios.
- Animate and/or simulate statecharts to further validate the scenarios.

16 Scenario Verification

As scenarios are the first artifacts created in the development cycle, they need not be verified against any documents or artifacts formerly created. Instead validation with the user stands central to the V&V process on this level. However, verification is not only to make sure the products of each development phase fulfill the requirements and conditions imposed by the previous phase, but also to make sure that requirements are complete and correct.

In validating the scenarios with the user, the specification is checked primarily for completeness, consistency and adequacy. Now the specification needs to be checked for internal consistency and completeness, for testability and to be free of ambiguity.

Internal consistency is checked by use of a glossary in which all the relevant terms are defined. Are all the events, the actions, the activities, all states, entities, data items, all conditions and constraints named, is their meaning defined, and are the same names and definitions used consistently throughout the document? Are all actors and all the scenarios named and defined? Do all the terms in the glossary describe distinct entities, that is: is any event, action, …, listed only once, or, if they are listed more than once having been named differently, are the equivalent terms marked as being synonyms?

Checking for internal completeness is achieved by asking questions like: are all external (and internal) events being handled, all inputs being processed and all outputs being generated? Are all actions and activities triggered by events defined? Are all data items and all resources that are needed to produce the desired results defined and determined? Check the user interface as far as it has been defined: can the user trigger all actions (need not be direct), are the system's reactions defined for all user actions?

Testability should be checked for at this time as well. Are the requirements quantifiable and measurable? Are data ranges specified, data formats defined, qualities quantified? Are expected results defined?

Finally, the specification has to be checked to demonstrate that it is unambiguous.

Many of these verification activities are supported in SCENT by the formalization of narrative scenarios in statecharts. By transforming natural language scenarios into statecharts, many of the problems of natural language may be alleviated, as ambiguities and inconsistencies may be found and vagueness is discovered.

17 Transformation of Natural Language Scenarios into Statecharts

Statecharts are an extension to the well known state-transition diagramming techniques. They help reduce the explosion in the number of states that occurs in state automatons modeling many parallel processes. Furthermore, they introduce the ability to decompose complex models hierarchically. Statecharts may be defined as state diagrams enhanced with the capacity to decompose complex models hierarchically and to model parallel processes [Har87]. They build upon synchronous communication deploying a kind of simultaneous broadcast communication[1].

The formalization step in SCENT is the transformation of structured natural language scenarios into a statechart representation. Statecharts developed from narrative scenarios may be seen as just another representation of the natural language scenarios created at first. Contrary to other approaches ([Fuc99, Som96]) 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. We define some helpful rules and guidelines, however, to help avoid some of the problems of natural language (e.g. ambiguity or vagueness. The guidelines are presented in section 12).

The creation of statecharts from narrative scenarios is a creative, innovative, in a way even artistic task that takes some experience. Designing good statecharts often depends on intuition and human genius. Creating statecharts out of scenarios is modeling, and, being a creative and innovative activity, the transformation step can not easily be formalized. Basili et al. indicate that most engineering methods are heuristic in nature rather than formal [Bas88]. This certainly still is true. We deem it appropriate then, to supply the developer with useful heuristics to accomplish the task at hand, that is, we rather provide heuristics on how to sensibly create statecharts from natural language scenarios than to try to formalize the transformation step.

Some helpful hints can be given to help developers transform natural language scenarios to statecharts. These heuristics are:

- As soon as first narrative scenarios have been developed (step 5 and 7 in the scenario creation procedure), create statecharts to match the scenarios. Even though some of the narrative scenarios will be altered and changed, and statecharts will have to be revised, adjusted and done over again (in parts at least), still the insight gained in developing them, pays the effort. They help to ask the right questions early in development, and often point to errors and misunderstandings.
- Create a statechart for each natural language scenario. If statecharts are getting to complex, make sure the underlying scenarios capture only one task each, check to make sure the scenarios are only aiming at reaching one specific goal. If necessary and appropriate (= corresponding to reality) divide scenarios and/or statecharts that are to complex by defining subtasks and subgoals to be reached.
- A single step in a narrative scenario usually translates into a state or a transition in a statechart[2]. As the steps are mapped to either states or transitions, missing states and transitions will emerge and need to be added. Superfluous states (and transitions, if any) need to be deleted or merged with needed states (or transitions).

1. All events are distributed to all active states in a statechart simultaneously. Events are neither saved nor queued, but they trigger a (one or more!) state transition immediately or they are lost.

- External (and internal – as appropriate) events are mapped to state transitions, the triggering event is the state transition to the initial state. External events that do not lead to a change in system state are irrelevant to the system – just discard them.
- Model the normal flow first. Integrate the alternative flows later on. Check if alternatives are missing. This can be done by investigating which events may occur in a given state. If an event could occur that has not been modeled, a transition is missing and usually an action or an alternative flow in a scenario is missing as well.
- The normal flow, all exceptional flows and all alternatives of a given scenario are captured in one statechart.
- Represent abstract scenarios as hierarchical statecharts. Abstract scenarios are modeled in a separate statechart, not in the statecharts they are part of. The resulting statechart that depicts an abstract scenario is plugged into its 'parent' statecharts at the appropriate positions.
- Statecharts are developed and refined along with the scenarios, thus providing for continual validation and ongoing check for consistency, omissions and ambiguities in the narrative scenarios. 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.
- Check the event list created for scenario elicitation to see if all relevant events are handled. Ask questions like: "The system being in this state, what will happen if the user does this or that?" Are states and events named expressively and consistently, following some scheme?
- Cross-check statecharts. Do states, transitions and events appearing in more than one scenario have the same names?
- Check the statecharts for internal completeness and consistency. Are all the necessary states specified? Are all the states in a statechart connected? Has every statechart an entry and an exit node? Are there no dangling links? Can every state be entered and left (except the final state)? Are all the necessary transitions specified? On what events can a state be left?
- Annotate statecharts with pre- and where appropriate with postconditions, with data ranges and data, and with non-functional requirements (the specifics and implications of this step / proposition are discussed in more depth in section 18).
- At first, statecharts are not integrated. These partial models help to enable traceability (from design and tests to requirements and vice versa). Statecharts may be integrated later on to create a model of the full system [Gli95] and to allow for further verification activities.

It is quite tempting to formalize the mapping from narrative scenarios to statecharts. However, we have found that most mapping schemes are too restrictive and that statecharts created by such schemes are not well-formed. We do define some guidelines on how a mapping of single steps in scenarios to elements in statecharts can be done, but we expressly point out and emphasize that these are only guidelines and no strict rules. A developer might violate these guidelines to create statecharts that are more readable, better understandable and more intuitive.

The guidelines are:

1. Decide in the beginning on the viewpoint you want to take in the model. The way statecharts are modeled depends quite radically on the viewpoint chosen. Most often a user-centered view is chosen, but at times a system-centered view might be just as appropriate. The closer to design and implementation the modeling gets, the more a system-centered description is reasonable, opportune and acceptable. However, for our purpose, a user-centered view is more appropriate, as in the origi-

2. If a working step in a scenario does not map to a state or transition, but needs to be modeled in more than one state or transition, respectively, this usually indicates that the step should be refined (broken down into substeps, its components). This will not normally be the case, as states can be created at various levels of abstraction and at various granularity levels – to the modelers will. But states at quite different abstraction levels in one statechart are an indication for insufficiently subdivided and refined scenario steps.

nal narrative scenarios the interactions between user and system are described in a user-centered way, and the formalization in statecharts, being just another scenario representation, should not digress to much from the narrative scenario description (backward-traceability, readability and understandability to users, use of 'statechart-scenarios' in validation activities, and so on). Thus we recommend to take a user-centered view in transforming narrative scenarios to statecharts.

What do we mean by the terms *user*- and *system-centered*, anyway?
In a user-centered view, the user plays the active part in using the system: Activities are named as they are beheld by a (fictive) user of the system (see Figure 19, model to the left). Thus, user activities are often modeled as states while in a system-centered view they are beheld as external events only and accordingly are modeled as transition events. In a user-centered view not only the triggering event and following system responses are modeled, but also system-external actions and activities are taken into account. Taking the concept a step further, not only direct system interactions – that is, only actions that directly lead to a system response – are considered and depicted, but system external events that lead indirectly to system relevant actions are also modeled in scenarios. For an example, consider the well-known ATM example: If the slot is obstructed, the fact that the card can not be inserted might well be modeled in a user-centered view, even though the actions to be taken in this case (user informs human teller or ATM administrator) are taken by the user and not the system. Even though this scenario outcome does not directly affect the system to be built, it is important in a user view of the system and it might lead to different design alternatives (e.g. install sensors/photo-electric barrier to detect the obstruction). Obviously, this approach leads to an enlarged context, and system boundaries are challenged.

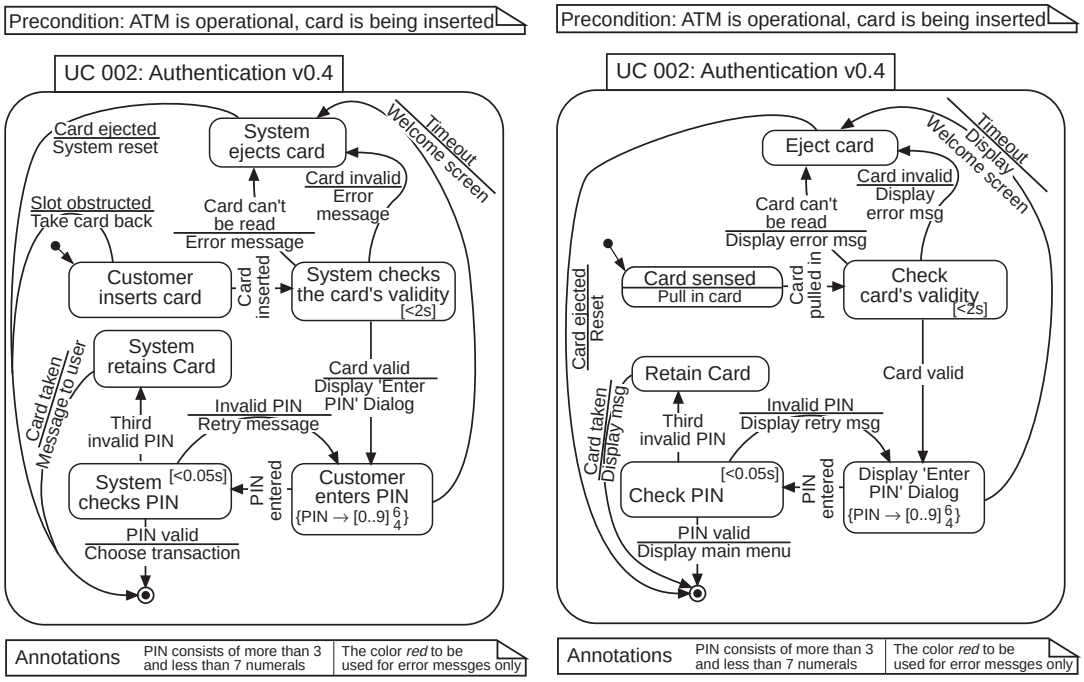


FIGURE 19. Different modeling viewpoints: user-centered versus system-centered

A system-centered view restricts the activities and actions to the ones done by the system, user actions are but mere external events. The system plays the active role and activities are named accordingly (see Figure 19, model to the right). To point out the difference once again: in a system-centered view, user actions normally show up as transition events only, while in a user-centered view, user actions (as well as system actions) may be mapped to states.

Naming is especially susceptible and dependent on the view the developer takes: The state named 'Customer enters PIN' in a user-centered view may be named 'Display *Enter PIN* -dialog' in a sys-

tem-centered view, ‘System checks PIN’ transforms into ‘Check PIN’ and ‘Choose transaction type (Withdrawal / Balance inquiry)’ is transformed into ‘Display main menu’ in a system-centered view.

We emphasize the fact, that neither the one nor the other is ‘the one and only right way’, but a user-centered view is more appropriate in the context we use scenarios in, in our method. But what is important, is, that once a view point has been chosen, the developer should stick to it throughout the whole development and use it consistently in all the models.

2. To map scenarios to statecharts, a ‘normal’ (that is: the intuitive and expected, natural) mapping of user actions to (external) events and of system actions to states or state actions is applied. Thus, user actions define external events which map to transition events, system actions map to transition actions or to states (see the following paragraph) The triggering event of the scenario maps to the transition to the initial state. Preconditions are depicted as conditions to the transition to the initial state. Decisions in the scenario (selections, ‘if … then …’ structures) map to conditional transitions.
3. Actions of the system may either be mapped to actions on transition, to states directly or to actions in states. The differences are quite subtle: An action on transition often is an action that can be performed in ‘nearly no time’, that is, the action can not be interrupted by other actions or events. Actions that take some time (usually called activities) are usually mapped to states. Furthermore, (short and lengthy) system actions may be grouped and mapped to actions in states: These are used to reduce model size and the number of states, preventing a state explosion and reducing the complexity of the model. Actions in states might be of different type (entry-, exit-actions, ‘do while in the state’-actions, application-specific actions triggered by conditions or events, and so on). All actions that do not lead to an externally perceptible change in state may be depicted as actions in states. Else extra states that just clutter the diagram need to be depicted. An example of actions often depicted in states are key stroke actions (only the Enter/Return key at the end of an input sequence leads to a change in state) or timer functionality (even though lapse of time is monitored, e.g. a count of ‘seconds passed since start of timer’ is kept, only when the desired time interval has elapsed a transition to the next state is triggered).

   There are no binding, obligatory rules, however. Actions on transition might well have duration, if the state transition is taken to mark the starting point of the action only. State actions always may be depicted as transition actions leading to an extra state. Again, it is not that important which approach is chosen, as it is to consciously pick the concept that best fits the purpose, and then stick to it. The SCENT-Method does not require one or the other approach, the choice is up to the developer of the statechart model.

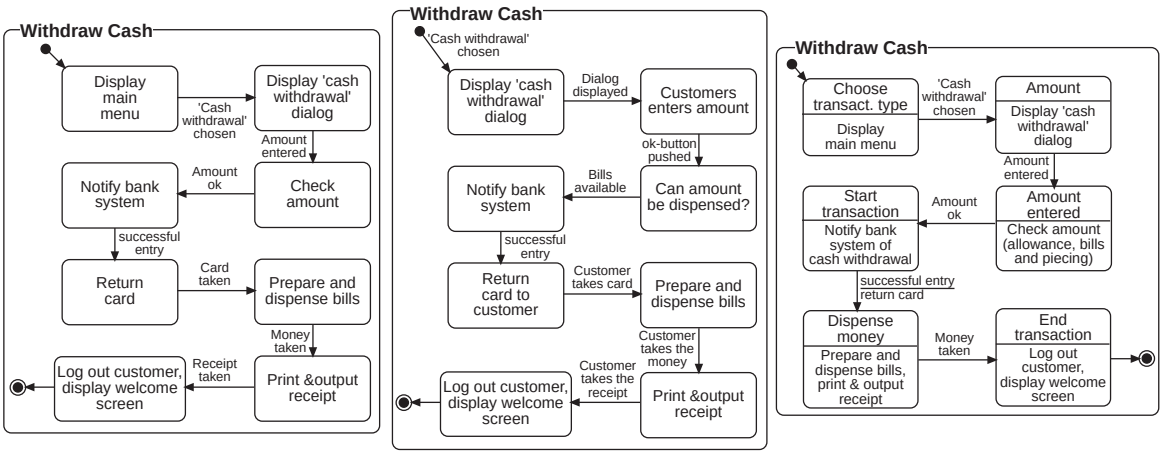


FIGURE 20. Different ways to model actions: on transition, as states and as actions in states

4. Pay attention to the fact, that user actions quite often are not atomic[3] events. This is especially true when user actions may be interrupted by system or other user events, as for example is the case in ‘user enters PIN’ (at any time the user may interrupt the procedure by pushing the cancel button. The event is not atomic as it may be decomposed on a lower level of abstraction to single key strokes.

The only relevant event on this level of abstraction is the user pushing the OK button, all key strokes of the numeric keys do not lead to a change of state on a high level of abstraction) or in ‘user inputs a card at an ATM’: The action of inserting a card is in reality – on a lower level of abstraction – a composed event, as it can be parted into the two events ‘sensing a card being entered’ [= the edge of the card has crossed a sensing device in the machine and the card now is (actively) pulled into the slot] as opposed to ‘the card is fully inserted’ [= the card has been pulled in completely].

If a user action is not an atomic event, the user action might well be mapped to a state, e.g. in the examples mentioned above, the action ‘Customer enters PIN’ and ‘Customer inserts card’ are mapped to states, with the following transition events: ‘PIN entered’ (could alternatively be named ‘OK button pressed‘) and ‘Card (fully...) inserted’ (see Figure 21).

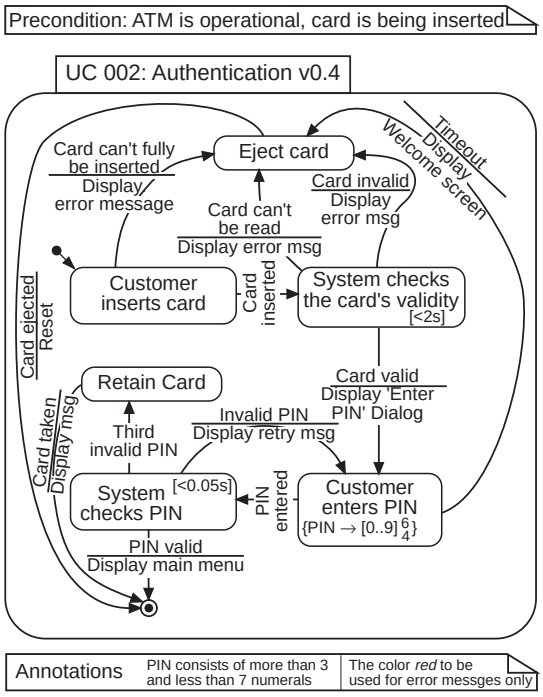


FIGURE 21. Authentication scenario in the ATM example

5. As has been mentioned before, each scenario is mapped to exactly one statechart at first. Thus, a scenario maps to a state (the statechart), and actions map to events, transitions and substates (internal states). Abstract scenarios represent substates that show up in other statecharts. That is why the developer has to make sure that abstract scenarios are closed, that is: that they do not have any transitions directly into or out of a substate. Abstract scenarios should be modular, that is compositional and plugable, without dependencies on other statecharts.
6. Alternative flows and exceptions establish new paths in the statechart. Alternatives and exceptions are mapped to alternative (conditional) transitions. The full scenario including all the alternative flows is modeled in one (probably hierarchical) statechart.
7. If desired the statecharts may be integrated into one system statechart [Gli95]. Be careful to name states and transitions, that is events, actions, consistently across statecharts (glossary!).

3. The notion of atomicity for events is not easy to define for it depends on the level of abstraction and detail. While one tenth of a second for a human may be considered to be atomic as a human can not react faster than that, it certainly is not indivisible / ‘uninterruptable’ for a modern computer.

   There are many more questions in details: Is a conditional event to be seen as an atom? Or is it non-atomic, as it obviously is a composite of a condition and an event? How about a composite event (e.g. pushing a modifier key with an alphanumeric key, double mouse click, key pressed vs. key depressed, key kept pressed, key released).

8. Preconditions have to be included in the statecharts. There are two ways to integrate preconditions in statecharts, namely in a banner-like notation and as conditions to the initial transition (section 18). If a given scenario has to be preceded by another scenario, then the other scenario may be included as the first state in the statechart to be developed. Abstract scenarios are to be included as a nested state and should not be mentioned in a banner only.

Creation of narrative scenarios and of statecharts is an iterative process. Normally, the first statecharts created of a system will have to be corrected, improved and revised. At first the developer verifies the statecharts against the specification, that is, against the original narrative scenarios. Then the statecharts have to be reviewed and validated by the customer and user (see section 20).

18 Statechart Annotation

If scenarios are to be used to create concrete test cases – specifying input values and expected output –, they need to include all relevant data and they need to be specified in detail. The information that is specified in the narrative scenarios needs to be included and represented in the statechart representation of scenarios also. However, statecharts just capture the behavioral aspect of a system. Data, quality requirements, object properties, user interface design and the like are not included in the model. Yet, as this kind of information is central for the development of test cases, we extend the statechart concept to include the information important for testing. In particular the additional testing information included in every statechart comprises the following:

1. Preconditions
2. Data: Input, Output, Ranges
3. non-functional requirements

The information is captured in annotations.

Preconditions

Preconditions are represented in statecharts as (part of) the initial state. Alternatively they can be shown as a part of the first state transition or in a banner-like notation (Figure 22, Figure 23). For example, if the precondition for a given scenario X reads: “To be performed only after scenario Y has been successfully executed.”, this may be modeled by assigning the compound event ‘initialEvent ∧ successfullExecutionScenarioY’ to the initial transition. This makes sense if preconditions are short (only one or two conditions) and if the conditions are easy to evaluate. In the case of extensive preconditions, it is often more comprehensible to annotate the statechart with a banner in which to describe the precondition (as is shown in Figure 23).

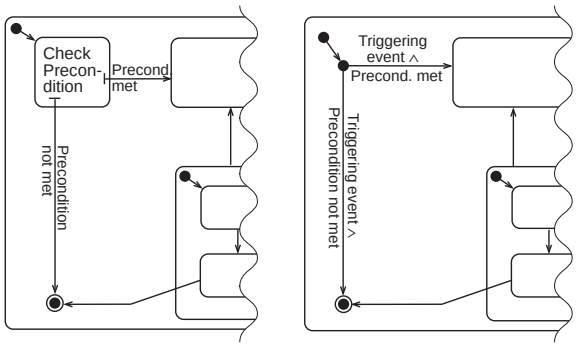


FIGURE 22. Preconditions modeled as conditional events in statecharts

Preconditions are capturing conditions that need to be fulfilled before the scenario can be performed. They can be classified to

- environmental conditions that need to be met (external state), and
- state of operation, that is, dependencies between different scenarios (internal state).

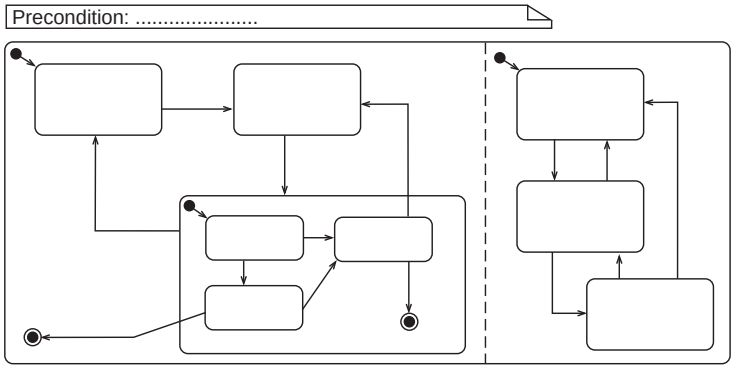


FIGURE 23. Precondition annotation in a statechart

The first category comprises conditions like operating system, or additional applications that need to be started before the scenario can take place (environment), disk space, system resources or memory needed to successfully go through the scenario (resources) or time constraints that need to be met (time restrictions). The second category encloses all dependencies between scenarios, e.g. that a certain scenario has to be run through before another can be performed, or that certain data has to be created/stored/evaluated… (in a given scenario) before the dependent scenario can be run. While external, environmental conditions usually can be specified quite easily and are quite stable over time, this is not true for internal conditions. They often are not easy to describe and to get right as they refer to internal state and build on given sequences of events. Thus, external conditions often will concern the whole system and will be specified in a special requirements document, while internal conditions are usually specified with the scenario that is concerned.

A special case of handling preconditions is the factoring out of common working steps in abstract scenarios: To help keep scenarios manageable and consistent it may be desirable to have scenarios of just a few single working steps that set up the needed environment to meet the preconditions of other scenarios. Thus, by executing the abstract scenario, the start state for (many) other scenarios is reached.

Preconditions determine the environment a scenario needs to be executed. They further describe the dependencies between the scenario to be executed and other scenarios. Thus, much of the information that is captured in dependency charts, is found in preconditions again. In fact, preconditions are just a partial, limited view of the dependency chart that belongs to the involved scenario. They are partial because they represent only a small fraction of all the dependencies depicted in a dependency chart, namely the dependencies that affect the one scenario that is represented in the statechart under consideration. And they are limited because they are concerned about and restricted to dependencies that are affecting the execution of the scenario under consideration, they do not, however, capture and describe the effects of executing the scenario.

Dependency charts and preconditions need to be kept consistent. After a dependency chart has been modified, the preconditions of all the affected statecharts/scenarios have to be checked, and vice versa.

Data

The second category of information that is captured in annotations in our approach are data, input and output. Data ranges have to be considered as well. It has to be noted that usually different ranges of

input data (partitions) lead to different behavior in the system and to different states in a statechart. Yet there are situations where data from different input partitions are represented in just one state. In this case it is mandatory, and in most other cases it is helpful to annotate the state transitions with data ranges. Thus, the test developer is guided in determining which values to take using extreme values (boundary value analysis).

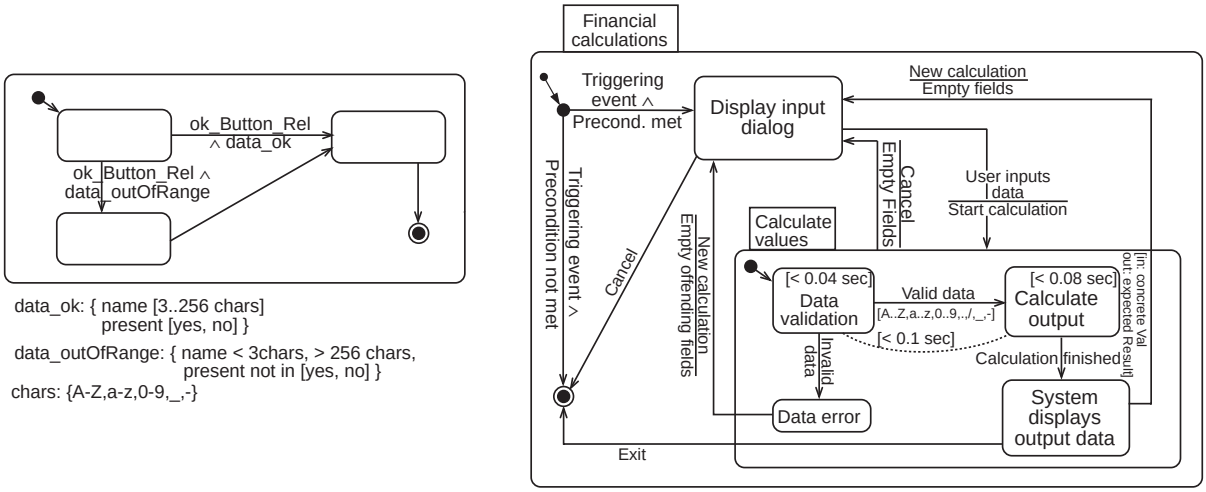


FIGURE 24. Data annotated in statecharts

It is quite often the case that results may be specified at the same time when input data is annotated. When ever possible annotate states that are computing values and outputting calculated data with the expected results for a couple key input values.

For exemplary annotations of data in statecharts, see Figure 24.

Non-Functional Requirements

The third information to be specified in annotations to statecharts are qualities (non-functional requirements). Scenarios mainly support the capturing of functional requirements. Qualities are not easily captured in scenarios and statecharts as

- They often cover or affect many scenarios or even the whole system (e.g. availability, reliability, robustness, portability, and so on. There are also other requirements, like the use of colors in all user screens, that affect most scenarios. Another example are performance requirements that span more than one scenario).
- They are hard to describe in the step procedure of scenarios as they do not nicely lend themselves to be specified in just a single working step of a scenario (in our ATM example it is not quite clear where to put the specification of what a password has to be like, for example: longer than 6, shorter than 13 characters, using at least two special characters, and so on. Should it be put to the first appearance of password in any scenario, to the first appearance in every scenario, to every appearance in any scenario? Should it be specified in a central document that captures all the qualities that affect the whole system and be referenced at all places where password appears in the scenarios?).
- They often are much easier to express by some other means than written natural language – e.g. in a picture, a screen-shot or a formula –, thus, scenarios should include diagrams, pictures and screen-shots. But are these means to be integrated into statecharts, and if so, how are they to be integrated?
- They are often forgotten and not specified when talking about a specific flow of actions or a process.

If the scenarios we create are to be the system specification, we need to include qualities and non-functional requirements. Some non-functional requirements (and even few qualities) may be annotated to states and transitions (e.g. performance requirements), most of them we just capture in notes.

Time, duration and other performance requirements (throughput, transfer rates, response times, …) are annotated in states (or alternatively with transitions. For an example of both, see Figure 25)

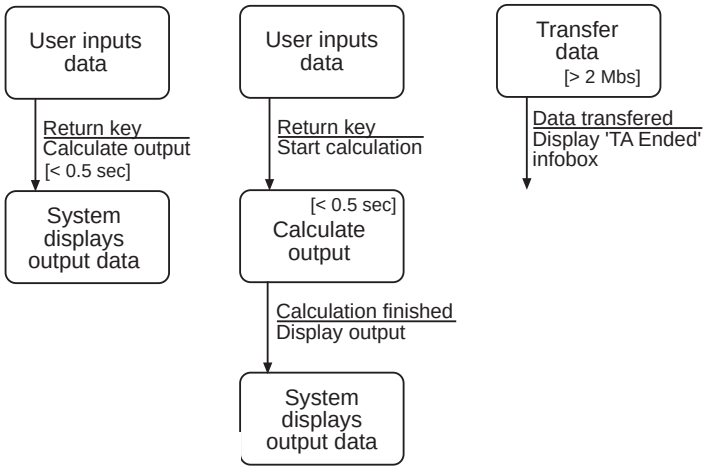


FIGURE 25. Performance requirements annotation

Time events can be specified as needed: as absolute events (‘at 10:00 AM every day’, ‘no later than 5:00 PM each last Friday every month’, ‘at midnight on Monday, March 1st 1999’) or as relative events (‘after event x has occurred’, ‘no earlier than 15 seconds after activity y has ended’). Time intervals and duration are defined by their start and end times, respectively. Thus, requirements like ‘if event z occurs three times in two minutes’ may easily be captured. More complex relations between events (e.g. complex events – conditional events, composite events and the like – or dependencies between events and other events, actions or activities) may be defined and used as appropriate.

Qualities/non-functional requirements that can be attributed to a specific scenario are specified in a special section of the scenario template (see appendix A). Data representation, views, screen layouts and masks are best specified in the appropriate scenario step and accompanied by graphics and pictures (e.g. ‘Display panel’, ‘Show dialog’, ‘User enters data’, ‘System monitors sensor’, …). In statecharts, qualities and non-functional requirements are annotated in a banner-like notation (Figure 21, Figure 26).

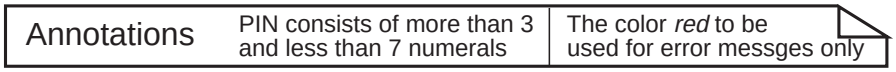


FIGURE 26. non-functional requirements annotation

Qualities that can not (easily) be attributed to a single scenario are specified in the system’s non-functional requirements specification. There, all constraints and restrictions imposed on the software and the development are collected. Examples are: memory requirements that apply to the whole system and at all times, interfaces to other systems that the system to be developed needs to interact with, legal restrictions and constraints, and so on.

19 Integration with Existing Methods

To guide and support the developer throughout the whole process of test planning, test case development and testing, the method needs to be tightly integrated with the general, commonly used software development process. This is achieved in our method by using scenarios: narrative scenarios, and statecharts as their formal representation. Scenarios are a means to elicit, capture and document requirements in many methodologies, notably also in the UML. In most projects, statecharts or other state automata representations will be developed during design anyway, so developing them early and at a quite abstract level – still taking a black-box view on the system – is only a limited extra effort, the expense of which can be justified by the verification and validation benefits gained.

Basic integration of the SCENT-Method in an existing development method (as for example the UML [Boo97, Rum99]) is easy and straightforward: Most of the modern methods do support some kind of scenarios and all of them feature some form of state-transition diagrams.

The more sublime integration is the (desired or even required) tool support for the management of scenarios, test cases, test sequences and test results, the management and integration of graphical models and textual specifications and traceability between models. Further integration problems arise as existing, tool-supported notations have to be extended (e.g. the statechart notation to specify performance and non-functional requirements, facilities to do path traversal in statecharts, and so on). Most tools allow to connect notes to most any model element, but there is no semantic to these annotations and thus tool-supported analysis of models is weak.

20 Drawbacks in Using Statecharts

The main problems and drawbacks in using statecharts to formalize natural language scenarios are:

- There is no formal way to convert natural language scenarios to statecharts, no automatic conversion is feasible.
- In the SCENT-Method, statecharts are derived from narrative scenarios. A statechart modeling one scenario is usually quite small and concise as compared to the statechart of a component or even a whole system. Thus, all paths of a statechart may be traversed to create test cases for system test (as is done in our method). But this only allows for a test of intra-scenario dependencies, not for a test of inter-scenario dependencies. All the relations and links between statecharts (and between scenarios alike) are neither modeled nor tested (save it be in dependency charts). Thus all dependencies, all connections and interactions between different scenarios would not be tested if we relied on test cases created by path traversal in statechart only. This means that all data dependencies (e.g. one scenario writing values, the other processing those selfsame values, a third scenario fetching and displaying these values), often also procedural dependencies (only after the initialization of scenario B may scenario A be executed more than once and concurrent to scenario C), and thereby the integration of different modules and packages, are not tested by test cases derived from statecharts only. That is why we introduce the notion of dependency charts and use them to enhance the test case suites that are created by path traversal in statecharts.
- Scenarios lead to a more transactional view, and are apt to test flows of actions from triggering event to final results. Thus, the method introduced in this paper helps to develop test cases to test an application end-to-end, in a process-oriented way, and is mainly suited to derive test cases for system tests. It has to be preceded by integration testing and extended by further test cases for system test, than the ones derived from scenario statecharts (domain testing, data-flow and transaction-flow testing).
- Statecharts represent another intermediate modeling step, as error prone and faulty as any other. The statecharts created from scenarios have to be validated with the customer even though the scenarios have been validated before. The formalization step introduces more (and new) faults into the system

model, but it is valuable as it helps to find contradictions, inconsistencies and omissions in the specification and helps to check a specification to be non-ambiguous. Thus more errors can be eliminated than new errors are introduced. But the (now formalized) scenarios, that is the statecharts, have to be carefully and thoroughly validated again.

CHAPTER 7 *Test Case Derivation*

The selection of appropriate test cases to test a system's conformance to a given requirements specification is an important issue in any serious software system development. In practice, the development of test cases is often unsystematic and the coverage of the requirements in system test is incomplete, as testing methodologies most often use code and input coverage only to determine the quality of testing.

If the specification of the system is given in the form of a finite state machine, there are a number of methods known to derive test cases and to create and refine a test suite for the system. One of these methods is shortly introduced in the following section.

Then the test case selection method of SCENT is presented. This method is easy to understand and apply as it relies on the notion of link coverage, only. Thus it meets our goal of being apt for and useful in practice and yet, as will be shown, to provide a systematic approach to create a first set of test cases that meets a given minimum test coverage criterion (transition coverage).

21 Test Generation Methods from Finite State Machines

In this section a method to derive test cases from finite state machines is presented. This method was chosen as a representative, for it is a generalization of some other methods and consequently stands for a big fraction of all finite-state machine testing methods. The method chosen is the W-method [Cho78]/ partial W-method [Fuj91].

The W-method introduced by T. Chow in his 1978 paper "Testing Software Design Modeled by Finite-State Machines" [Cho78] is restricted to tests of the control structure of a software product, the control structure being modeled by a finite state machine. It is called "automata theoretic" in his paper as it is based on results in automata theory. The test sequences developed by the method are guaranteed to reveal any errors in the control structure, if certain restrictions are meet. The method is concerned with the global control structure only, as opposed to the modeling and implementation of operations and the resulting control structure in particular. Next operation and next state depend solely on current state and the given input. The assumptions the method relies on are: The machine is specified completely, it is minimal, a fixed initial state is given and every state is reachable.

In applying the method, the maximum number of states in a correct design is estimated at first, then test sequences based on the existing design are generated and finally the outputs are verified. The main part (and contribution) of the method is the generation of test sequences: Test sequences are created by first constructing two sets of sequences, the sets P and Z. P comprises all sequences that force the state

machine from the initial state in a desired state by any path to this state. Z is the union of all inputs (the input alphabet) concatenated with the characterization set, the inputs in the concatenations raised up to the n^{th} power where n denotes the number of estimated state count minus actual state count. A characterization set consists of all the input sequences that can distinguish between the behaviors of every pair of states in the minimal automaton. The test sequences are formed by concatenation of sequences in P with sequences in Z. The method is correct if requirements and design machine accept the same inputs (input alphabets are the same) and the estimate of the maximum number of states is correct.

In the partial W-method [Fuj91] the W-method is refined by applying a two phase process and restricting the characterization set to a subset, built by the union of characterizing state sets called identification sets of the states. In the first phase of the approach, the implementation is checked to include each state defined in the specification. The second phase checks all the remaining transitions defined by the specification not tested in the first phase yet for correct output and transfer. As does the W-method, the partial W-method requires a reset function (to be implemented correctly in the application under test), the finite-state machine of the specification to be minimal, completely specified and deterministic, the number of states in the specification and implementation not to differ by more than a given n and all the states to be reachable. For details we refer the reader to [Fuj91].

There are some assumptions in applying these methods that are quite questionable in practice. The method is shown to reveal all output and all transfer errors and even find all missing or extra states if the specified and the implemented state machine do not differ by more than a few states[1]. But in practice it is often the case that they differ by many states, often even by a factor of n times the number of expected states in the specification (see [Bei95]), exposing hidden finite state behavior that leads to ”parallel universes in hyperspace” [Bei95]. Furthermore, as we are talking about black box testing, it can not be guaranteed that the implementation will not have more states than the specified ones plus a few.

Another problem arising in this method is the assumption that a reset function is given and that this function is implemented correctly, that is, any time and from any state you can return to the initial state[2].

Thirdly, a restriction that is quite stringent for statecharts is that the state machine has to be minimal, completely specified and strongly connected. Statecharts, used in our approach as a semi-formal representation of scenarios, are seldom completely specified. Often they are not even minimal. However, there are algorithms to minimize state machines, as well as to check if they are completely specified and if not, to complete their specification enforcing a reaction to any input of the input alphabet in any state.

The different classes of errors given in [Cho78] are helpful to compare the fault detection power of different approaches. Chow differentiates between operation, transfer and extra state errors. Given two minimal finite-state machines, they can differ in:

1. The outputs created. The outputs created by one machine differ from the ones created by the other machine, but the difference can be removed (that is the two machines are made equivalent) by changing only the output function of one machine.
2. The state reached by a given input sequence. The state reached is different in the two machines, but they can be made equivalent by changing only the next-state function of one machine (but without adding or deleting states).
3. The number of states. Since the two finite-state machines are minimal, they can not be equivalent, having a different number of states. The two machines can be made equivalent by adding states to one of them.

1. This limitation is a practical, not a theoretical limitation: As the size of the test suite increases exponentially as the number of extra states increases, it is infeasible and impracticable to deploy the method if difference is more than, say, 3 to 4 states.
2. This means returning to the ”true” initial state and not to the initial state in a parallel universe. To achieve this goal it is often necessary to cold-start or, if available, to reinitialize the system so as to ensure that no unwanted or unexpected state – information, data, … – is left in memory, influencing any consecutive run of the program.

or in any combination of the former.

For a test method to have full fault-detection power means that it covers all errors of the above mentioned categories or any combinations thereof.

The presented approach guarantees that – provided the two automata have the same input alphabet, the estimate of the maximum number of states implied by the specification is correct and all the above mentioned assumptions are met – all sequencing errors in the design will be detected, that is all operations, all transfer and even extra state errors up to a certain bound are detected. Thus, these methods prove very powerful, but also quite time consuming and consequently expensive for all the analysis that has to be done to deploy them and the big number of test cases generated by them.

As this overhead and the effort put into this kind of testing do not correspond to a proportional increase in the quality of the resulting product, nor assure the software to be error-free (as only a partial abstract high-level model is tested), it is often not appropriate to do a full fault-detection power test. Accordingly we take a limited testing approach in our method.

In SCENT the generation of test cases is done in a three phase process, the three phases being

1. Preliminary test case definition and test preparation during scenario creation
2. Test case generation from statecharts and from dependency charts
3. Test set refinement by application dependent strategies (intuitive, experience-based testing)

The following sections introduce and further explain each of these mentioned steps.

22 Test Case Generation During the Creation of Scenarios

During the creation of scenarios, the developer documents testing that has to be done by either specifying (abstract) test cases directly or by hinting at and describing test data, boundary values, exceptions and situations that have to be tested especially thoroughly or that take special attention (see the scenario creation process in chapter 4). This constant focus on testing during analysis, specification and design helps improve testing directly and indirectly, as many errors are uncovered in the early phases of the development when test cases are being prepared.

The test cases created at the time of scenario creation need not be precise and detailed (i.e. they need not specify the exact input and the expected output yet), but serve to capture ideas and give hints what has to be tested – and probably also a short note on the why and possibly on the how. This helps to arrive at a complete set of test cases as the scenarios are multiple times read, reviewed and validated. Test cases that might slip otherwise are documented early in the development cycle, at the time the developer recognizes their importance. While scenarios are being refined, so are the test cases (or at least the hints and ideas that help the tester later on to specify test cases).

A considerable advantage of this approach is that test case generation is started early in the development process. Therefore, testing is freed at least partially of time pressure and tight schedule occurring in the last phases of the development process. Furthermore, the developer is confronted with testing issues before design and implementation and thereby is reminded to design and implement for testability as well.

23 Test Case Generation from State- and from Dependency Charts

The second phase to obtain a first set of test cases is the derivation of test cases from statecharts and from dependency charts.

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 the paths that are chosen according to a given criterion in a finite state machine 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 and thereby fulfilling the minimum mandatory testing requirement for graphs (node and link coverage). This link cover is simple and can be performed at acceptable expense either manually or automatically.

Basic path testing according to McCabe [Mcc76] is another possible choice. A more elaborate coverage could be chosen as desired (e.g. switch or n-switch coverage [Cho78, Gon70, Pim76] 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.

Dependency charts are valuable in testing as they support the derivation of additional test cases, thus enhancing the test suite developed by paths traversal in statecharts. The tests derived from dependency charts are very important as they ensure that dependencies between scenarios are tested. By deriving test cases from scenarios (formalized in statecharts), the intra-scenario behavior and relations are tested. By deriving test cases from dependency charts, the inter-scenario relations and dependencies are tested.

Test case derivation from dependency charts is done – as can be expected in a graph – by traversing path in the graph structure. 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. The dependency chart is especially useful to specify and test for unwanted behavior: try to break constraints and restrictions (e.g. take a loop more often than allowed, execute a scenario without first executing a preceding scenario, perform a scenario without having the necessary resources prepared by another scenario).

However, the derivation of test cases from dependency charts is somewhat more problematic and complicated than the derivation of test cases from statecharts. While in statecharts test cases are created by path traversal in the chart and extended by taking into account the data and performance annotations in the statechart, in dependency charts the various kinds of dependencies have to be considered. Timing dependencies that are specified quite easily might be very hard to test. Conditions that have to be satisfied to test certain dependencies might be extremely expensive to establish or simulate. Furthermore, as dependency charts get more complex, there is a multitude of paths that may be tested (combinatorial explosion[3]) as all different dependencies between scenarios have to be tested. In general the tester will

3. This certainly is true for statecharts also. But while in a statechart every node is but a system state, in dependency charts every node is a full scenario with many states, transitions and thus with many possible exceptions and alternatives. Furthermore, scenarios (and the statecharts depicting them) are restricted to a small part of the system, while a dependency chart is a model of the whole system, depicting dependencies system-wide and spanning many scenarios.

have to restrict the number of test cases by prioritizing tests and choosing paths and conditions that have to be tested.

24 Refinement of the Test Suite

The test suite developed so far has to be refined and enhanced by further test cases to ensure high quality testing. Further test cases are developed by appropriate testing methods. Include loop and queue testing[4], synchronization tests, stress and load test, special GUI and database tests, device testing and so on. The extension is done based on experience and existing test suites, it might well be supported by existing testing approaches as for example the approaches described in [Bei95]. As supporting methods to create further test cases either data-flow or transaction-flow testing can be chosen. In data-flow testing all input, all output and all predicates should be covered by generated test cases. A full link coverage may not have been achieved as internal (data-flow) nodes may not be modeled in the statecharts. Extended equivalence partition testing, boundary value analysis and heuristics can be used to determine further test cases. The test cases specified or initialized at scenario creation are, as has been mentioned before, part of this extension to the initial test suite.

As scenarios are prioritized, the customer or developer may decide that high priority scenarios are tested more rigidly, applying another coverage criterion than link coverage to the statecharts to derive further test cases. Refined test case derivation may be based on priority assignment by customer.

Another valuable asset to expand and refine the initial set of test cases created by path traversal in statecharts is the tester's experience, intuition and knowledge of the most common errors and faults as well as the tester's knowledge of error distributions.

4. It has to be noted that loops are not that common in scenarios. Mostly scenarios are linear, include some branching, but only seldom loops. The transactional structure of scenarios picturing a sequential flow of actions accounts for this property. If loops exist, they are usually simple and localized [Bei95] as for example retry loops after detecting a data input error.

CHAPTER 8 *Related Work*

Even though literature on scenarios abounds, test case derivation from scenarios is yet in its infancy. Scenarios are used in most object-oriented development methods, notably in the UML (Unified Modeling Language) also, and many different approaches have been developed over the last decade. Yet, in the area of testing, only few scenario-based approaches exist. In the following, we review some approaches with regard to scenario creation, scenario formalization and support for testing activities:

1. Jacobson's Use Case approach [Jac92, Jac94a, Jac94b] was one of the first to disseminate the use of scenarios and propagate a user centered requirements capture and specification. The approach does not, however, propose any defined procedure on how to create scenarios, nor on how to use scenarios in testing. Scenarios are not formalized in any way. No specific description format or template is advocated. Furthermore, use cases/scenarios – as groupings or collections of functionalities and requirements - are rarely truly independent in practice. But in Jacobson's work only very limited support for modeling dependencies between scenarios is given (uses and extends relations). However, to derive tests from scenarios the dependencies between scenarios have to be known, else crucial parts of the system will or may not be tested.

   A well-known feature of Jacobson's approach are the so-called overview diagrams, an example of which is given on page 94 in appendix C.

2. Firesmith [Fir94] extends the scenario approach to model scenario life-cycles, constraints and relationships between different scenarios. Furthermore, scenarios are not necessarily triggered by an actor, thus allowing for internal scenarios. A level concept for scenarios is introduced. The benefit of scenarios in testing is only hinted at, the development of test cases is not addressed at all.

3. Regnell et al. [Reg95, Reg96] in their approach aim at overcoming the problem of lacking synthesis of single scenarios to reach a full picture of the whole system by formalizing and integrating the use cases of a system. Testing is touched upon, but no strategy to test case selection is defined. The main aim of [Reg95] is to present improvements to the OOSE/Use Case Driven Analysis UCDA approach of Jacobson [Jac92] by identifying weaknesses and problems in UCDA and defining a possible solution. [Reg96] focuses on the representation, extending the former approach to include a hierarchically structured representation. Testing and the derivation of test cases is not an issue in the approach.

4. Potts et al. [Pot94] describe a scenario analysis approach with an emphasis on an inquiry-based procedure in the analysis process. They define a process for capturing and describing discussions about and rationale of requirements. Interesting is in their approach that they take changing requirements and the reasoning process into account, supplying a model for scenario evolution. Testing as such is not an issue in their paper.

5. Lee et al. [Lee98] use Petri nets for the analysis and integration of scenarios. They emphasize the importance of incremental specification of partial system behavior and of consistency and complete-

ness checks for requirements engineering techniques. Then they argue that an extended Petri net approach satisfies these demands. They define Constraints-based Modular Petri Nets CMPNs, introduce Petri net slices to analyze system behavior, present a procedure to create CMPNs from scenarios and show how the model can be checked for consistency and completeness. As do some of the approaches mentioned above, this approach tries to integrate and analyze scenarios, it does not however define a procedure for test case derivation from scenarios.

6. Message sequence charts-based scenario modeling: Message sequence charts MSCs are used in different approaches to formalize scenarios or to capture system requirements (see [And95, Reg95, Reg96] for examples). Yet MSCs suffer from the disadvantage that they are getting overloaded fast when all exceptions and alternatives of a scenario are to be integrated in one chart. On the other hand, they miss an abstraction mechanism to decompose coherent descriptions.
7. Scenario modeling based on activity diagrams: Activity diagrams (which recently have been added to the UML) are a variant of state diagrams, but they are used for a work- and control-flow representation of the system [Rum99]. Transitions in activity diagrams are triggered by termination of actions or activities that are represented in states. Activity diagrams are used for the formalization of use cases/scenarios [Sch98] as activity diagrams are regarded as a means well suited to communicate with the customer. However, in our opinion activity diagrams are not appropriate for modeling scenarios, as activity diagrams allow for ‘spaghetti control structures’ as did old-fashioned flow diagrams. Furthermore, activity diagrams are not intuitive: The flow constructs have been extended by different concepts (e.g. swimlanes to represent responsibilities, modeling of concurrent constructs, inclusion of objects) and thus they are not easily understood. And finally, the semantics of activity diagrams are in comparison to the semantics of state machines and state transition diagrams, quite weak (timing model, activity state construct).
8. Hsia et al. [Hsi94] have proposed and described a more formalized approach to scenario creation and validation, constructing scenario trees. Their approach is based on regular grammars and equivalent (conceptual) state machines. Scenario creation and formalization are core-parts of the approach. The use of scenarios to derive test cases however is only shortly sketched and no further procedure has been specified. Their main use of scenarios in testing is for acceptance test. Our approach is close to the one proposed by Hsia in many respects. It differs though in central issues:
    1. Scenario elicitation is conducted via scenario trees in the Hsia approach. In our approach we define an iterative step-by-step procedure to create scenarios in structured natural language.
    2. Scenario trees are formalized into regular grammars in the Hsia approach. These grammars describe a conceptual state machine, defining a formal abstract model. One model for each user view is created. The definition, use and maintenance of these grammars is labor-intensive, requires special training and skills on the side of the developers and the grammars are not intelligible to customers and users. Changes to scenarios reflected in the grammars are quite cumbersome.
       In contrast we formalize scenarios into statecharts, every statechart representing one scenario. Statecharts need not be (but may be!) integrated. The notation is expressive and understandable to users, and changes in scenarios are easily reflected in the statechart-representation.
    3. In the SCENT-Method a definite procedure for test case derivation from statecharts is specified, creating concrete test cases (defining the settings/environment and the input values needed for test execution). In the Hsia approach basis paths are used to generate scenarios from the conceptual state machine, these scenarios are used as input for acceptance testing. No concrete test cases are created.
    4. Statecharts in SCENT are annotated with preconditions, data and non-functional requirements to enhance the creation of test cases. No equivalent concept has been defined in the Hsia approach.
    5. Dependencies and relationships among scenarios may in SCENT be modeled in dependency charts and dependencies may be tested for accordingly. In their approach, Hsia et al. do not propose any way to handle inter-scenario dependencies.

To our knowledge no scenario-statechart-based methodology for test case derivation does exist. SCENT uses structured natural language scenarios to capture user requirements and create a behavioral and functional system description. Then extended statecharts are used to formalize the scenarios. A primary set of test cases is generated from statecharts by traversing paths in the statecharts. Test suites are improved by using information relevant for test case creation collected during analysis and scenario creation. The method presented in this paper is novel with respect to the synthesis of the afore mentioned factors into one single process.

CHAPTER 9 *Experience Report*

In this chapter, a short summarizing report of the experiences made in applying the approach in two projects at ABB Corporate Research Center in Baden-Daettwil/Switzerland is given. Then the most important findings are summarized.

The projects that were chosen to apply the method presented in this report to and to validate the method with were applications for remote monitoring of embedded systems in high voltage power transformation and distribution. The first system was to monitor substations in power distributing systems (bays in a high voltage busbar system), the second to monitor transformers. System fundamentals are described in some details in [Its98].

The developers in the projects were experienced in object-oriented methods and had a vague idea of scenarios and their use in software engineering. Some of them had used an unstructured scenario-approach before. However, experiences were quite ambivalent as different developers had quite different perceptions of what scenarios are and what they are used for. Furthermore, in most projects scenarios had not been kept up to date, so the use of scenarios in testing was new to all of them. To introduce the SCENT-Method, the scenario creation procedure was presented to the developers in a half-day workshop. The formalization of narrative scenarios to statecharts was explained and illustrated in a brief example, but mostly the transformation step was left up to the developer. We provided them with the heuristics given in section 17. Annotating statecharts was explained in detail, as was test case development by mere path traversal to achieve link coverage. The development of extended test cases by use of boundary-value and dirty testing, testing for qualities and non-functional requirements and further testing was left to the developers and testers as it is mostly experience-based.

In the following sections we first describe the application of the method in two projects, and then we summarize the findings that are most noteworthy.

25 Application of the SCENT-Method in Two Projects in Practice

In the first project (the high voltage installation monitoring project) scenarios were created after a more conservative requirements specification had been written. The scenarios were developed based on the existing requirements specification as it was considered to take too much effort to redo the whole specification again. So users were not directly involved in creating scenarios, but were asked to validate the scenarios created from the existing requirements specification. In the old specification, requirements were listed, classified and assigned to distinct categories for functional requirements and for non-func-

tional requirements (interface, performance). Thus, requirements were listed and bundled, not according to their belonging to a work-flow, to a task or a business process, but according to a requirements classification scheme.

Creating scenarios from the specification turned out to be quite challenging because of this difference. In accordance to the scenario creation procedure used in the SCENT-Method, first the actors and system external events were determined. Then coarse scenarios were developed based on the specification and task description of the system. 25 scenarios resulted from this step. A first validation with domain experts and users showed quite a few discrepancies and inconsistencies. The duties and tasks of some actors (=roles some user plays) had been misinterpreted. A major revision of the scenarios was indicated, especially the different users roles and responsibilities had to be redefined. Even a new actor had to be introduced to the model. Thus, validation of the first scenarios resulted in a major rework on many scenarios. They were refined and abstract scenarios were factored out to total 33 user scenarios.

In a next step, further scenarios were developed representing the engineering view of a system engineer tailoring the generic system to fit the specifics of a given installation. The new scenarios were to capture and document the requirements a system engineer demands of the monitoring system environment to customize and install the monitoring system. This led to the creation of what we called developer scenarios. As no written requirements specification existed, these scenarios had to be created from scratch. So the full-blown scenario creation procedure was first time deployed with direct user involvement. A total of 12 scenarios was created in the system engineering view. The existing requirements were integrated and reflected in the new scenarios where appropriate. Throughout the whole process the scenarios were checked for consistency with the existing specification.

After a first internal review the scenarios were partially[1] validated by users by walking through them. Some misunderstandings were discovered and the scenarios adapted.

Then the scenarios were formalized and synthesized into statecharts. The statecharts were partially[2] integrated into a system statechart. The metamorphosis from scenarios to statecharts proved to be a creative step that could not be formalized in an easy and straightforward manner. A lot of experience, creative thinking and problem solving is needed to create good, well-formed statecharts. The hints given in this paper on how to transform informal natural language scenarios into formal statecharts try to capture the experiences we made while creating statecharts from scenarios in this project.

A prototype implementation was developed based on the conservative requirements specification and first test cases were developed. Because of the restricted functionality of the prototype and as the project was postponed, the scenarios were not extensively used to derive test cases. More especially, test cases were not developed systematically to cover all system functionality, but testing was restricted to show the main features to be functional and guarantee for an operational prototype to be used for demonstration purposes only. The high voltage installation monitoring system thus served to show that the scenario creation and transformation procedures of the SCENT-Method were appropriately applicable and useful in practice.

In a second project – a transformer monitoring system – the test generation capabilities of the method were to be applied, and requirements coverage and test quality were to be determined. First, a requirements specification for the system had to be written. This time, a scenario approach was chosen and the SCENT-Method was applied from the beginning. The requirements of the new system were elicited and documented in 14 scenarios. These scenarios were created in accordance to the step-by-step procedure described in this paper. Following the SCENT-procedure the scenarios were then transformed into state-

1. All developer scenarios were validated by a installation designer, and the most important user scenarios were validated by user representatives. The rest of the user scenarios was not validated by users, but only verified against the existing traditional requirements specification.
2. Only user scenarios were integrated, while developer scenarios were left ‘stand alone’.

charts. A test specification was developed. From the statecharts test cases were derived by path traversal. Test case derivation was refined and elaborated by including black-box test strategies to develop test cases missed by pure path traversal in statecharts. Test cases to cover dependencies between scenarios were derived. Meanwhile a prototype of the system was developed and validated with the scenarios. Finally the prototypical system was tested according to the test specification. The aptness of the approach was confirmed by collecting user feedback and by a requirements coverage evaluation.

26 Findings

In this section, we summarize some of the findings and experiences made by applying the method. The statements are mostly summaries of customer and developer feedback. Feedback was sought for actively and played a major role in developing and enhancing the method. An evaluation of the results achieved by the method, more especially of the quality of the test suites generated by applying the method, by means of using quantitative measures, has not been done so far. The coverage (most appropriately, it would be a requirements coverage in this case) reached by generated test suites has not been thoroughly measured yet. Some preliminary measurements suggest that a 100% requirements coverage may not be reached without further 'manual' test case development, that is, extended tests for qualities and non-functional requirements have to be developed, and test suites generated by path traversal in statecharts have to be enhanced by dirty testing (as is part of the test generation procedure in the SCENT-Method). The development of the needed additional test cases however is not systematic, but mostly intuitive and experience-based.

The central findings are:

- Positive customer feedback: The use of scenarios was perceived by the developers as helpful and valuable in modeling user interaction with the system. Scenarios were found to be well-suited to capture a user's view of the system. Developers especially appreciated the integration of the users in the development process and the direct feedback they received in creating and refining scenarios. Customers appreciated scenarios as they were understandable and clearly outlined normal (expected) behavior.
- Iterative Approach: Developers appreciated the iterative character of scenario creation and refinement and the activities of scenario formalization that run in parallel to the creation of narrative scenarios. The iterative procedure helped to accommodate frequent validation activities and allowed for a smooth integration of changed requirements. As reported by Weidenhaupt et al. in an exploratory survey of practice [Wei98], scenario definition is not a one-time activity, but scenarios evolve over time. From the four types of evolution mentioned in [Wei98] – top-down decomposition, from black- to white-box scenarios, from informal to formal scenario descriptions and incremental scenario development – three are facilitated in the SCENT-Method:
  1. A top-down decomposition is done by first creating overview scenarios (e.g. at the level of business processes) which are then refined to scenarios capturing the coarse flow of actions (e.g. task level scenarios). These 'task-level' scenarios are further refined to a sequence of atomic actions (scenarios on the level of activities). Top-down decomposition is central for a method to be applied in practice, as observation of current practice shows that the majority of stakeholders in the requirements specification process prefer to develop scenarios in a 'top down manner' [Rol98].
  2. Scenarios in SCENT are at first recorded in free natural language, then they are structured and formatted into a scenario description according to a template, and finally they are formalized into statecharts. Thus the evolution type 'from informal to formal' is covered.
  3. Incremental scenario development is supported by the iterative refinement procedure defined in SCENT.

  The evolution from black- to white-box scenarios is not enforced by the SCENT-Method, but it is easily integrated if required or desired, as internal scenarios can be included anytime.

- The scenario creation procedure: The guidance that is given by the procedure on how to create scenarios, what to do first, how to start and which way to proceed, was received very well by the developers. This might be partially due to the experiences made in former projects, where scenarios were developed without a guiding procedure and without any training for the developers. In these projects, developers had some problems to use scenarios effectively.
- The scenario template: The scenario creation process is nicely supported by the scenario template. However, there are two sides to the template. On the one hand, the scenario template is useful and welcome as a means to structure the scenarios. It establishes a common layout and provides a structure for content, thus supporting the developer in keeping the scenario representations consistent. Furthermore, it is helpful in assuring that all important facets are covered and nothing is left out. On the other hand, the template has to be concise and short, lest developers will be reluctant to use it. The effort needed to write and update the scenarios has to be kept in close bounds. At first we created an elaborate, extensive template to describe scenarios. Later on, we realized that it was not handy to be used in real projects. So we simplified the template to the form presented in this report.
- Use of natural language: In SCENT, we use structured, but unrestricted natural language scenarios, because we found that formalizing the scenario language would be an impediment to user involvement and to developers' engagement and commitment to use scenarios [Arn98]. Natural language has many advantages and disadvantages (see chapter 5). Still we think that the advantages of using natural language scenarios outweigh the disadvantages, more especially so, because one of the most prominent features of scenario-approaches is the improved user involvement – and this advantage would be lost if formal languages were used.
- Systematic test case development: The systematic approach to test case development proved helpful in improving testing. Test case creation was not problematic as the chosen link coverage in statecharts is simple, yet powerful. Moreover, the test case generation procedure can easily be tailored to meet project-specific needs.

The application of the method (and of scenario approaches in general) was not without difficulties, however. Some of the problems that surfaced in applying the method were:

- Management of scenarios: Scenario management was considered a major problem throughout the development. Scenarios are at first natural language descriptions and as such, they are not easily kept consistent. Furthermore, appropriate tool support is missing. As scenarios interrelate with many of the other artifacts produced in the software engineering process, there is threefold to manage: Keep scenarios consistent in themselves (as one specific scenario may exist in many versions and representations [e.g. as a natural language scenario and as a statechart] and as different scenarios may be related one to another), keep scenarios and other documents and artifacts consistent, and keep scenarios up-to-date when requirements, the environment and the developer's understanding of the problem are changing.
- Granularity of scenarios is a major concern: It is not easy to know when to stop refining a scenario neither is it clear when a scenario has the right depth to capture all the needed user requirements. This is a general problem not restricted to the scenario approach. It boils down to the problem of what is still a part of the specification and what is design and implementation [Swa82].
- Scenarios are not suited for a model of system internas: Contrary to the positive experience of modeling user interaction in narrative scenarios, it was troublesome to model system internal interaction with descriptive scenarios. The developers preferred more formal and concise models (e. g. MSCs, state machines, and the like) to accomplish this task. More than once developers stated that they would use scenarios only to describe the user interaction part of a system, but not the more technical parts which play an important part in the domain of embedded systems in which we applied the approach. So, most likely, in many projects, there will not be a pure scenario-based specification, but a mixture of traditional and scenario-based specification.
- Need for detailed scenarios to be useful in testing: Developers were reluctant to specify scenarios down to the level of detail needed for test case derivation. If scenarios are to be used for test case generation, scenarios have to be complete with respect to alternative flows and exception handling, and data as well as performance requirements have to be specified and included in the scenarios.

The problems mentioned above are not specific to the SCENT-Method, but they show, to some extent, in any method that is based on scenarios.

The formalization process in SCENT also posed some problems, because the mapping of actions in natural language scenarios to states or transitions in statecharts 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 narrative scenario by another developer. The transformation can not easily be formalized and automated. Most states, transitions between states and especially most events are mentioned implicitly or explicitly in narrative scenarios. Yet there is no direct mapping between scenarios and statecharts: states and transitions need to be determined, actions in narrative scenarios need to be mapped to either transitions or states, alternative courses need to be integrated into the statechart that depicts the normal flow of actions also, and abstract/extended scenarios may or may not be integrated in the same statechart.

Furthermore, the integration of qualities and non-functional requirements in scenarios and in statecharts is problematic (except performance requirements, as they are easily and closely integrated in statecharts as state or transition constraints). Qualities may be described in narrative scenarios, but often that leads to redundancy and inconsistencies as qualities often affect or concern more than one scenario (see section 18). Merely annotating statecharts with banners is helpful, but not an optimal solution, as it prevents automatic test case development. Test case generation from statecharts thus is rendered a manual task again, that only can be supported by appropriate tools.

Finally, the extended testing relies heavily on problem understanding, domain knowledge, experience and intuition. An improved integration with existing testing strategies and alternative testing methods [Bei95, Kan93] to generate the enhanced test cases is desirable.

Despite the drawbacks and problems we encountered, the experiences gained and results achieved are quite promising as the main goal of the method, namely to supply test developers with a practical and systematic way to derive test cases to create a first set of test cases, has been reached.

CHAPTER 10 *Conclusions*

27 Achievements

The main goals of the SCENT-Project were:

1. To develop a method to support specification-based testing that is embedded in the software engineering process,
2. To support the tester in systematically developing test cases for system test,
3. To (re)use artifacts from earlier phases of the development process in testing to utilize synergies between the different phases,
4. To integrate the development of test cases in early phases of the development process to alleviate time-pressure in testing.

First experiences show the scenario-based approach described in this report to be quite promising. The goals mentioned above have been reached at least in parts, as will be shown in some more detail in the following.

The first goal really consists of two parts: a method for specification-based testing and the integration of this method with 'normal' software development methods. SCENT is specification-based – or rather scenario-based, thus being requirements-based – as analysis scenarios that are formalized in statecharts are used to derive test cases. The functional system specification that is captured in scenarios is used to guide testing. SCENT as opposed to other approaches does not require a formal language to be used to specify the system and thus avoids many of the problems and cost that formal languages carry. Yet the disadvantages of using natural language as a specification language – vagueness, ambiguity, hard to check for consistency and completeness, hard to automate or support with tools – are reduced by the transformation to statecharts.

The approach is clearly useful for functional testing, while for structural testing it will be applicable in parts only as a full state automaton model of the system (including all internals) would have to be provided.

The embedding of the method in the software engineering process is shown in chapters 4, 6 and 7. The intertwining of the software process and the test specification process is shown in Figure 27.

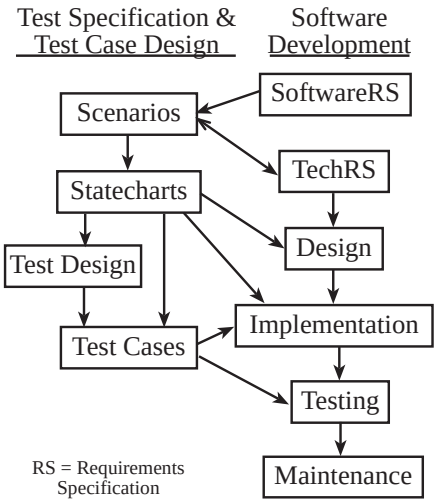


FIGURE 27. Embedding of the method in the software development process

The integration of the method with normal software development methods was addressed in section 19.

The second goal (systematic test case development) is reached by traversing paths in statecharts. Thereby, test cases for valid input sequences or chains of events are generated. At the same time, tests for invalid sequences must be created. By use of a graph to model the system, systematic test case development for functional properties is achieved, and by using conditions, data and performance annotations, further refined functional tests are created and tests to check for performance and required qualities are developed. In applying the method in practice, it proved helpful to generate a first set of test cases. The generated tests had to be extended by more dirty testing, though, and tests for non-functional requirements had to be added. Functional requirements were fully covered by generated tests however.

The third goal (reuse of early artifacts in later phases) is reached by using analysis scenarios for testing again. The cost of doing so is dealt with in the next section.

By using the method, we find that test cases are developed early in the software development process which helps to alleviate the problem of testing done under immense time-pressure at the end of the development only. Furthermore, we find that designing test cases while still analyzing, designing and implementing the system helps to find or avoid many faults that otherwise would only be found during testing.

28 Cost of the Approach

The cost of the approach is twofold. First, natural language scenarios have to be developed. This is an extra cost only if development would not be done with scenarios otherwise. Obviously, the approach makes sense in projects that deploy scenario-based requirements elicitation, specification and development. If only alternative (traditional) specification techniques are used, then the approach as presented in this report still has some nice advantages (improved user involvement, systematic test case development and so on), but it probably will not be the most cost-effective solution.

Secondly, representing scenarios with statecharts requires some work to be done that otherwise wouldn't. But the extra work put into 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 systematic 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. In our experience the benefits outweigh the additional cost by far.

29 Conclusions

In this report we have presented the SCENT-Method, a scenario-based approach that helps improve testing by advocating systematic test case development. 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. Furthermore, we have introduced a diagram to capture dependencies between scenarios and we have sketched how to make use of this dependency graph to further improve systematic development of test cases.

The method presented in this paper is novel with respect to

1. the definite scenario creation procedure, a feature that is missing in many scenario approaches currently in use,
2. the guidelines for using natural language to describe scenarios – notably the rules on the use of specific language constructs –, supporting the developer in using natural language more effective and precise which is very important if natural language is to be used as a specification language,
3. given heuristics for guiding the transformation and formalization of narrative scenarios to statecharts – statecharts being a more formal representation of the scenarios, that further help avoid some of the shortcomings of natural language,
4. an extension of the statechart concept to include information important for testing (data, preconditions, qualities and non-functional requirements),
5. using scenario statecharts to derive test cases from,
6. and, over all, the integration and synthesis of the aforementioned aspects into a single method and process.

The SCENT-Method has been applied in practice to two projects at ABB in Baden/Switzerland (see chapter 9). First experiences are quite promising as the main goal of the method, namely to supply test developers with a practical and systematic way to derive test cases, has been reached. The approach of using formalized scenarios as a means to support systematic test case development proved helpful.

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 main findings are (for more details see chapter 9):

1. Scenario use was perceived to be helpful and valuable by the developers as well as by the customer and end-users. Scenarios proved to be very well suited to model user interaction, they were, however (and not surprisingly), quite cumbersome to model system internals. Internals were modeled preferably by statecharts directly.
2. The management of narrative scenarios was the major problem throughout the development process (to a minor extent this is true for formalized scenarios as well).
3. The formalization process is not unequivocal and definite, which may be seen as a problem, as a statechart developed from the same narrative scenario by different developers may differ significantly.
4. The generation of a first set of system test cases is greatly improved by the systematic approach taken in the SCENT-Method.

In applying the method, we made the experience that many of the problems found are of a general nature (e.g. scenario management is really the problem of managing any collection of natural language descriptions or specifications; it is a problem of document management aggravated by the huge amount of dependencies and interrelations between the documents). A similar statement could be made for the use of natural language as a specification language.

Leaving these general problems aside, some specific issues concerning the SCENT-Method remain. As has been pointed out, a method without a tool to support it is quite limited. This is true for SCENT as well. However, we don't want to develop and present yet another testing tool that is a stand-alone solution. Rather we strive for an integration of the method with existing tools. Further research has to be done in this respect.

The integration of non-functional requirements with scenarios and statecharts is another issue that needs to be addressed. Up to date, scenarios are normally mere descriptions of system behavior, at best enriched with some additional information, as for example non-functional requirements and qualities. A scheme of how qualities are to be captured, represented and consistently managed in a scenario approach does not exist. Qualities often span more than one scenario – thus it does not make much sense to annotate the qualities with every scenario that is affected –, yet just having a global repository that does not have any connection to the scenarios save it be some reference to the scenarios, is not an optimal solution, either. Our research will lead in the direction of a tighter integration of scenarios and non-functional requirements.

Another issue is consistency. As more than one representation of scenarios is created, namely the narrative and the statechart-based representation, the two have to be kept consistent. And more than that, the scenarios need to be in line with design models and implementation as well. There is no ready-made solution to this problem, one of the obstacles being the fact that scenarios are unstructured natural language descriptions. Related to the mentioned problem, is the need for a concept on how to uniquely identify requirements in scenarios, relate scenarios to statecharts and keep the requirements traceable to the statecharts and following models as well. This really is the general problem of relations between different models: How is the code connected to the design models, how is the design model tied to the analysis model, how can requirements be traced to solutions, specification to code? How are different models composed, for example, how are classes in a class diagram tied to scenarios? How are classes identified in scenarios?

And finally, further research is needed to formalize the use of data annotations and performance requirements in deriving test cases from annotated statecharts. In the SCENT-Method the process still relies heavily on experience and intuition of the tester. Even though this facet never will be fully eliminated from the testing and test development process, the dependency from tester's skills and brilliance has to be reduced. This best might be achieved by a definite, formal and thus repeatable procedure. We plan to cover this facet in future research.

References

[And95] M. Andersson, J. Bergstand, "Formalizing Use Cases with Message Sequence Charts," Master thesis, Department of Communication Systems at Lund Institute of Technology: Lund, 1995.

[Arn98] M. Arnold, M. Erdmann, M. Glinz, P. Haumer, R. Knoll, B. Paech, K. Pohl, J. Ryser, R. Studer, K. Weidenhaupt, "Survey on the Scenario Use in Twelve Selected Industrial Projects," GI Fachgruppe 2.1.6 Requirements Engineering, Aachener Informatik-Berichte 98-07, June 1998.
The publication may be downloaded from the ftp-server of the Rheinisch-Westfälischen Technischen Hochschule of Aachen/Germany as technical report 98-07 (ftp.informatik.rwth-aachen.de/pub/reports/1998/index.html).

[Bas88] V. R. Basili, H.D. Rombach, "The TAME Project: Towards Improvement-Oriented Software Environments," IEEE Transactions on Software Engineering, vol. 14, # 6, pp. 758-773, 1988.

[Bei90] B. Beizer, Software Testing Techniques, Second Edition ed. New York: Van Nostrand Reinhold, 1990.

[Bei95] B. Beizer, Black-Box Testing, Techniques for Functional Testing of Software and Systems. New York: John Wiley & Sons, 1995.

[Ben92] K. M. Benner, S. Feather, W.L. Johnson, L.A. Zorman, "Utilising Scenarios in the Software Development Process," IFIP WG 8.1 Working Conference on Information Systems Development Process, 1992.

[Boe81] B. Boehm, Software Engineering Economics. Englewood Cliffs, N.J.: Prentice Hall, 1981.

[Boo97] G. Booch, I. Jacobson, J. Rumbaugh, The Unified Modeling Language. Santa Clara: Rational Software Corporation, 1997.

[Cho78] T. S. Chow, "Testing Software Design Modeled by Finite-State Machines," IEEE Transactions on Software Engineering, vol. 4, # 3, pp. 178-187, 1978.

[Coc96] A. Cockburn, "Basic Use Case Template," Humans and Technology, Technical Report 96.03, Salt Lake City 1996.

[Coc97] A. Cockburn, "Using Goal-Based Use Cases," Journal of Object-Oriented Programming, vol. 10, # 7, pp. 56-62, 1997.

[Fil98] D. Filippidou, "Designing with Scenarios: A Critical Review of Current Research and Practice," Requirements Engineering Journal, vol. 3, # 1, pp. 1-22, 1998.

[Fir94] D. C. Firesmith, "Modeling the Dynamic Behavior of Systems, Mechanisms and Classes with Scenarios," Report on Object Analysis and Design, vol. 1, # 2, pp. 32-36,47, 1994.

[Fuj91] S. Fujiwara, G. v. Bochmann, F. Khendek, M. Amalou, A Ghedamsi, "Test Selection Based on Finite State Models," IEEE Transactions on Software Engineering, vol. 17, # 6, pp. 591-603, 1991.

[Gli95] M. Glinz, "An Integrated Formal Method of Scenarios Based on Statecharts," in: W. Schäfer, P. Botella (eds.) Software Engineering - ESEC '95. Proceedings of the 5th European Software Engineering Conference, Sitges, Spain. Berlin: Springer, Lecture Notes in Computer Science 989, pp. 254-271, 1995.

[Gon70] G. Gonenc, "A Method for the Design of Fault-detection Experiments," IEEE Transactions on Computers, vol. C-19, pp. 551-558, 1970.

[Har87] D. Harel, "Statecharts: A Visual Formalism for Complex Systems," Science of Computer Programming, vol. 8, pp. 231-274, 1987.

[Hol90] H. Holbrook, "A Scenario-Based Methodology for Conducting Requirements Elicitation," ACM Software Engineering Notes, vol. 15, # 1, pp. 95-104, 1990.

[Hsi94] P. Hsia, J. Samuel, J. Gao, D. Kung, Y. Toyoshima, C. Chen, "Formal Approach to Scenario Analysis," IEEE Software, vol. 11, # 2, pp. 33-41, 1994.

[IEE90] IEEE, Standard Glossary of Software Engineering Terminology. IEEE Std 610.12-1990: IEEE Computer Society Press, 1990.

[Its98] R. Itschner, C. Pommerell, M. Rutishauser, "GLASS: Remote Monitoring of Embedded Systems in Power Engineering," IEEE Internet Computing, vol. 2, # 3, pp. 46-52, 1998.

[Jac92] I. Jacobson, M. Christerson, P. Jonsson, G. Övergaard, Object Oriented Software Engineering: A Use Case Driven Approach. Amsterdam: Addison-Wesley, 1992.

[Jac94a] I. Jacobson, "Basic Use Case Modeling," Report on Object Analysis and Design, vol. 1, # 2, pp. 15-19, 1994.

[Jac94b] I. Jacobson, "Basic Use Case Modeling (cont.)," Report on Object Analysis and Design, vol. 1, # 3, pp. 7-9, 1994.

[Jac95a] I. Jacobson, "Formalizing Use-Case Modeling," Journal of Object-Oriented Programming, vol. 8, # 3, pp. 10-14, 1995.

[Jac95b] I. Jacobson, M. Christerson, "A Growing Consensus on Use Cases," Journal of Object-Oriented Programming, vol. 8, # 1, pp. 15-19, 1995.

[Kan93] C. Kaner, J. Falk, H.Q. Nguyen, Testing Computer Software, second ed. New York: Van Nostrand Reinhold, 1993.

[Lee98] W. J. Lee, S.D. Cha, Y.R. Kwon, "Integration and Analysis of Use Cases Using Modular Petri Nets in Requirements Engineering," IEEE Transactions on Software Engineering, vol. 24, # 12, pp. 1115-1130, 1998.

[Man99] P. Mandl-Striegnitz, H. Lichter, "Defizite im Software-Projektmanagement - Erfahrungen aus einer industriellen Studie," (“Deficiencies in Software Project Management – Experiences made in an industrial study”) in: Informatik/Informatique, Magazine of the Swiss Informatics Societies, vol. 5, 1999, pp. 4-9.

[Mcc76] T. McCabe, "A Software Complexity Measure," IEEE Transactions on Software Engineering, vol. 2, # 6, pp. 308-320, 1976.

[Mye79] G. J. Myers, The Art of Software Testing. New York: John Wiley & Sons, 1979.

[Pim76] 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.

[Pot94] C. Potts, K. Takahashi, A.I. Anton, "Inquiry-based Requirements Analysis," IEEE Software, vol. 11, # 2, pp. 21-32, 1994.

[Reg95] B. Regnell, K. Kimbler, A. Wesslén, "Improving the Use Case Driven Approach to Requirements Engineering," Proceedings of the 2nd International Symposium on Requirements Engineering, York, England, 1995.

[Reg96] B. Regnell, M. Andersson, J. Bergstrand, "A Hierarchical Use Case Model with Graphical Representation," IEEE International Symposium and Workshop on Engineering of Computer-Based Systems, Friedrichshafen, 1996.

[Rol98] C. Rolland, C. Souveyet, C. Ben Achour, "Guiding Goal Modeling Using Scenarios," IEEE Transactions on Software Engineering, vol. 24, # 12, pp. 1055-1071, 1998.

[Rum99] J. Rumbaugh, I. Jacobson, G. Booch, The Unified Modeling Language Reference Manual. Reading, Mass.: Addison-Wesley, 1999.

[Rys98] 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, Technical Report 98.12, Zurich, 1998.

[Sch98] G. Schneider, J.P. Winters, Applying Use Cases: A Practical Guide. Reading. Mass.: Addison-Wesley, 1998.

[Som96] 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.

[Spe94] I. Spence, C. Meudec, "Generation of Software Tests from Specifications," SQM'94 Second Conference on Software Quality Management, Edinburgh, Scotland, UK, 1994.

[Sut98] A. Sutcliffe, "Scenario-Based Requirements Analysis," Requirements Engineering Journal, vol. 3, # 1, pp. 48-65, 1998.

[Swa82] W. Swartout, R. Balzer, "On the Inevitable Intertwining of Specification and Implementation," Communications of the ACM, vol. 25, # 7, pp. 438-440, 1982.

[Wei98] K. Weidenhaupt, K. Pohl, M. Jarke, P. Haumer, "Scenarios in System Development: Current Practice," IEEE Software, vol. 15, # 2, pp. 34-45, 1998.

[Yam98] T. Yamaura, "How to Design Practical Test Cases," IEEE Software, vol. 15, # 6, pp. 30-36, 1998.

Appendices

In the appendices further information and examples on the following topics are given:

Appendix A: The Scenario Template. The template for describing scenarios as used in the SCENT-Method, including short descriptions of the expected contents of each section, is presented. Additionally, a document template is specified. The document template is a wrapper for the individual scenarios.

Appendix B: A description of a specific scenario using the scenario template, commenting on and illustrating the use of the template.

Appendix C: An example system is described in excerpts to illustrate the scenario creation and formalization procedure of SCENT. The generation of test cases from formalized scenarios is briefly illustrated by developing some test cases from the example application.

Appendix D: To exemplify the use of dependency charts we give a short example (a small part of the well-known library system is used) that illustrates how dependencies between scenarios are depicted in and how test sequences are developed from dependency charts.

Appendix E: An overview of the notation used in dependency charts is given.

Appendix A: The Scenario Template

The scenario template provides a pattern for describing scenarios. It defines the required contents and structures of scenario descriptions and determines the layout. In SCENT, scenarios are created using this template. However, the template may be tailored to specific needs, e.g. sections may be added or extended – say a section to record rationale and source of requirements, or a section to describe dependencies, may be added –, but none of the predefined sections should be dropped (loss of essential information).

The scenario template is part of a general document template (see the ‘Scenario Documentation’-Template on page 86) that serves for capturing information about the scenario documentation in general. While this information is recorded once for all scenarios, the scenario template is repeatedly used to describe every single scenario. In particular, the general documentation contains an overview of all scenarios and actors. Furthermore, it includes the dependency chart(s) thus also showing the dependencies between scenarios.

Scenario Template

Scenario Description

Descriptive, Characteristic Scenario Information

Scenario <Identification/Number> <Name>

(*Name* is preferably a short active verb phrase describing the goal to be reached by and the purpose of the scenario, respectively)

Description, Purpose, Goal of Scenario in Global Context

<Scenario description stating the purpose and/or the goals to be reached by the scenario, level of detail as needed>

Actors

<Name of the role of the main actors, may be extended by a short description of the actors responsibilities>

Precondition

<Process states and conditions that must be met before the scenario can be performed; state of system before the scenario is executed; scenarios that must be executed before this scenario is entered>

Postconditions

Success: <State of the system after the scenario is performed successfully>

Failure: <State after scenario traversal if goal can not be achieved>

Triggering Event

<Event or action triggering the scenarios execution: time events allowed>

Normal Flow, Success Scenario

<Describes the flow of actions normally taken in the scenario in a step-by-step procedure>

<Numbering> <Description of working steps in short active verb phrases>

Alternative Course of Action, Exceptions, Error Handling

<Listing of all alternative flows of actions in the scenario. Alternative flows may be included in to the normal flow of actions - if properly marked and if they comprise but one or two alternative actions>

<Conditional step of the main scenario>: <Action/Activity>

<Conditional step of the main scenario>: <Action/Activity>

Further Information

Performance and Non-Functional Requirements

<Time and run-time constraints, time dependencies, (user) interface requirements, …>

Version

<Continuous version number >

Change History

<History of major changes from preceding version>

Owner

<who created this scenario, who is in charge of keeping it up to date>

Type

<normal, abstr*act, extending*>

Connection to and Integration with Other Model Elements/Models

<Links to diagrams, tables and graphics belonging to, specifying or clarifying this scenario, especially links to statecharts and MSCs, that were created from this scenario have to be listed>

Open Questions/Known Problems

<List of open questions about this scenario, the “To-Be-Determined”s, any decisions not decided on yet. Might well include a list of possible variants, that later on will lead to extensions to this scenario, or to alternative flows and exceptions, respectively>

Test Planning, Test Cases

<Test cases evident at the time of scenario creation, remarks to test planning, testing strategies and procedures. Description of requirements: who brought in a specific requirement, change history of requirements, the rationale of requirements and decisions, that is, why certain decisions were taken the way they were, and the like. Any information important to test the system, instructions and reminders with regard to system test>

The individual scenarios that are described according to the template presented above, are compiled in a document for which we specify the following template:

Document Template: Scenario Documentation

Name of Owner
Business Unit
Address
Telephone / Fax
Email

Project Data

Name of Project:
Customer, Place:
Business Unit:

Document Data

Name of Document:
Date (of this version):
Status: <*working paper, approved, final*>
Total Number of Pages: 3
Document / Change History: <*changes to the document*>
Documents Referenced: <*what documents are referenced*>

Distribution

Name	Company	Department	Copies	Remarks

Abstract

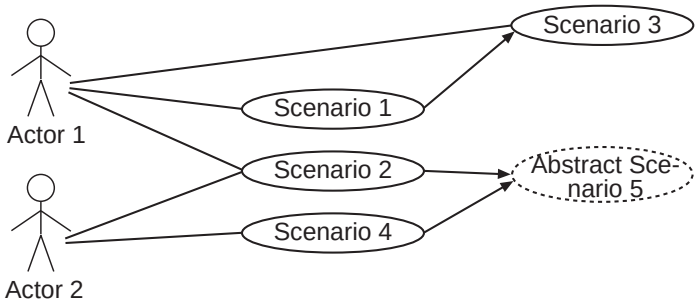
Short description of the system and of the contents of the paper.

Table of Contents

Overview Diagram

<if desired, include a graphical overview diagram in UML notation, depicting the actors, the main scenarios and the relations between them>

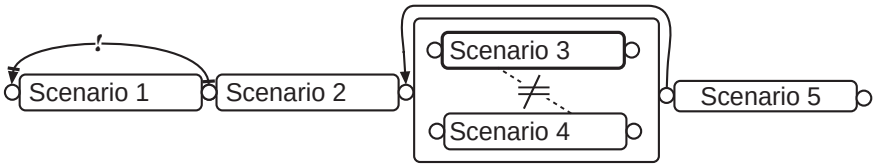
Notation:



Dependency Charts

<include all the dependency charts created in the project. Dependency charts are used to capture the relations, connections and dependencies between scenarios. Temporal, causal and abstraction dependencies are depicted, including data, call and time dependencies>

Notation:



List of Actors

Name of actor (role)	Short description of the actor	Type of interface (UI, API, files, …)

List of Scenarios

Name of scenario	ID	Short description of the scenario	Page	Pre-condition	Post-condition	Actors involved

Appendix B: An Example of Using the Scenario Template

In this section we illustrate the use of the scenario template by specifying a scenario in some detail. We provide explanations and hints as appropriate to support users in applying the template. The scenario chosen is a user setting the room temperature in a building equipped with a building automation system (*the system* in the following scenario description).

The example is commented. Comments appear outside the frames. Everything inside the frames would actually be included in the scenario description as a continuous text (except the alternative flow that is listed twice to illustrate alternative representations of exceptional flows).

Scenario Description

Characteristic Scenario Information

Scenario RO2: User sets desired room temperature

Description, Purpose, Goal of Scenario in Global Context

The user adjusts the room's thermostat setting to the desired temperature. The system raises or lowers the temperature to the specified level.

Could just as well have been only one sentence: The system adjusts the room temperature to the level desired by the user. *The extent and detail of this description depends on preferences of user and/or developer and on business culture and documentation standards.*

Actors

occupant (of the room)

Precondition

-

Postconditions

Success: room temperature is within half an hour in range of ±0.5°C of desired temperature

Failure: desired temperature can not be achieved, system informs responsible custodian

Triggering Event

change in preferred temperature setting for a specific room as desired by occupant

No precondition is specified even though the system has to be running as a prerequisite to the scenario described. Furthermore, a thermostat has to be installed, the thermostat has to be monitored by the system, communication between system and thermostat has to function properly, the room's occupant must be able to adjust the thermostat and similar conditions could be specified. However, in describing scenarios, initial conditions that have to be true for all or many scenarios can be specified once as an initialization scenario that has to be run before any other scenario (or before specific scenarios). Thus it is not necessary to repeat the same conditions in every scenario which might give rise to problems in keeping scenarios consistent.

Postconditions are to specify the successful end of the scenario as well as failure conditions (what will be the state of the system in case of failure?). All exceptions and all alternative flows of actions should lead to either a successful end or to one of the defined failure states.

The triggering event could also be specified as:

Occupant sets thermostat to desired temperature. *Or:* Room is too warm or too cold. *Better even is a specific event name that is defined in the glossary (e.g. setTemp, or TemperaturChangeRequest). The triggering event is to be specified for scenarios triggered by an actor (external event) and for internal scenarios (internal event triggering a scenario).*

In writing down the flow of actions, the triggering event may constitute the first action (as is done in this example) or the system reaction to the triggering event may be seen as the first action. Which of the two representations is chosen depends mainly on preferences, but it should be used consistently throughout the specification.

There are a couple of possible ways to capture the normal flow and all the alternatives. To illustrate some of the possibilities, we include two descriptions of the different flows: first, normal flow and alternative flows are separated, in the second example the flows are interlaced.

Normal Flow, Success Scenario

1. The occupant adjusts the thermostat of the room to the desired temperature.
2. The system observes the changed value within 30 seconds (polling frequency), does a validity check and saves the value to the database.
3. The system compares the actual room temperature to the desired room temperature.
4. The system adjusts the temperature (execute scenario 12 “Adjust room temperature”).

Instead of another scenario being called, we might well have specified all the necessary steps right here. Yet it is efficient to factor out behavioral sequences that are used in more than one scenario as it reduces paperwork and helps keep scenarios consistent. The next working steps (now factored out in the called scenario) might have been:

4. The system calculates the needed increase/decrease in heating water through-put and/or needed air conditioning to heat up/cool the room to the desired temperature in less then half an hour.
5. The system checks if through-put can be increased/decreased to the desired level without cutting other heating requests (execute scenario 10 “Calculate through-put”).
6. The system adjusts the valve in the desired room to increase/decrease the heating water through-put to the desired level and/or increases/decreases the level of air conditioning.
7. As soon as the temperature in the room reaches …
8. …

This description reveals one of the difficulties of using scenarios to specify a system: the developer easily stumbles into the pitfall of using scenarios to merely functionally decompose the system. However, scenarios at this level and for the intended purpose should be descriptions of expected behavior in response to an external event or a user input, that might use and affect many of the functions of a system.

The example also illustrates another point.There are two normal flows resulting from the "User sets desired room temperature" scenario, namely the "Room temperature is too low" and the "Room temperature is too high" scenarios, not considering the case of the room temperature being just fine, as in this case, the occupant will not adjust the thermostat. Because of this, all the actions that follow the step of comparing the actual to the desired temperature, and thus of determining whether the room has to be heated or cooled, have two actions (see above): either increasing or decreasing heating water through-put and/or increase/decrease the level of air-conditioning. This makes the formulations quite unwieldy and, more than that, adds unnecessarily to the complexity and decreases understandability of the scenario. Therefore we strongly suggest to separate the two cases of high and low temperature and describe them in separate scenarios. These scenarios would be called then in the scenario presented above. Thus the scenario would read:

4. Temperature is too high: Room is cooled to desired temperature (execute scenario 14 "Room temperature is too high")

   Temperature is too low: Room is heated to desired temperature (execute scenario 13 "Room temperature is too low")

Furthermore, the factoring out of these scenarios helps to reduce consistency problems, as the mentioned scenarios will be used more than once (room temperature needs to be adjusted not just because of users requesting a temperature change, but also because of environmental conditions – e.g. season, daytime, outside temperature, position of blinds).

Alternative Course of Action, Exceptions, Error Handling

1a. The thermostat can not be adjusted (defective).
   1a.1 [expected user behavior] The occupant of the room informs the responsible custodian of the defect. Exit scenario.

2a. The thermostat can not be polled, the actual room temperature can not be retrieved
   2a.1 The system tries three more times to poll the thermostat, waiting 30 seconds in-between the calls.
   2a.2 **if not** (the problem persists), **then** return to step 2,
   else the system checks for communication line error and for sensor defects (execute scenario 37 "Check for defects").
   2a.3 **if not** ((the problem persists) **or** (defects were detected)), **then** return to step 2,
   else the system prints out an error and status report for the tests that were run, logs the incident (specifying the sensor and cause for the entry) in an error log and informs the responsible custodian. Exit scenario.

2b. The thermostat returns a value out of range (validity check failed)
   2b.1 The system polls the thermostat again
   2b.2 **if** the problem persists (validity check failed again),
   then the system informs the responsible custodian, exit scenario
   else return to step 2.

4a. Actual temperature is equal to desired temperature.
   4a.1 No actions need to be taken. Exit scenario.

If not stated otherwise a scenario is continued at the step following the point of interruption after an alternative flow of actions has been taken. In the example, if the thermostat can not be polled at first, (exception 2a.), but in a second or third call the desired values can be fetched (successful step 2a.1), scenario flow control will return to step 3 in the main scenario. If control is to return to a specific working step in the normal flow (not the one following the step that caused the exception), then the last step in

the alternative flow has to specify the step in the normal flow which to return to. Thus, if the normal flow should be resumed at the step causing the exception after the alternative actions are finished, this has to be specified explicitly by a return to the step in the normal flow (see step 2b.2, else branch, in the example above). Moreover, if the scenario has to be terminated after an alternative flow has been executed, this has to be specified by an exit statement (see example above)

Note that no action/further working step ***must*** *be specified in an alternative flow (4a in the example above): explicitly mark this case by specifying a 'no action' to be taken, exit the scenario or jump to the step in the main scenario where you want to resume execution.*

Notice the numbering scheme used in the alternatives section: The first number relates to the number of the working step in the normal flow scenario where the alternative or exception catches, the following letter is used to distinguish different alternatives or exceptions and the final number is a continuous count of the alternative working steps.

Further take note of alternative 1a. The problem of a thermostat that can not be adjusted lies outside the scope of the system (that is, the defective thermostat – e.g. thermostat knob can not be turned anymore, probably because knob is broken – can not be detected by the system). Yet we may choose to specify the exception in the scenario, to document expected, presupposed or required user behavior, or to initiate further thought as to different system designs that allow to handle the exception. The issue might be taken up in the open questions section of the scenario description.

The level of abstraction and detail in describing scenarios may vary widely. In the example, many more details might be defined and documented. For example step 2 might read alternatively on a lower level of abstraction:

2. The system polls the thermostat for the actual room temperature.
3. The thermostat reports the actual temperature.
4. The system polls the thermostat for the desired room temperature.
5. The thermostat reports the desired room temperature.
6. The system validates the desired room temperature according to …

And so on.

It is quite obvious that the level of detail may vary widely. Normal flow and alternatives may be specified just on a very abstract level or they may define the working steps to insignificant minutiae. It is the responsibility of the developers and part of the art of specifying a system to find the right level of detail, so as not to get lost in a glut of trifling irrelevancies, but not to omit any significant and substantial requirements either.

The level of detail needs to be in correspondence to the purpose of the scenario. If scenarios are to serve as a (functional) system specification all user requirements need to be captured. If users specify a special function how to validate the desired temperature (e.g. temperature must be in the range of 15<t<24), then the function should be specified in a scenario step. If users do not care how validation of input is done, but rather they care that values are checked, then specifying system validates value is enough. If it is just technical reasons that require validation of values, the validation step might not show at all in the scenario description intended for system specification. It will show, however, if the scenarios are used as basis for design and implementation as well.

Another way to describe the normal and the alternative flows is by integrating alternative flows in the normal flow, marking alternatives by indent and key words. As an example we present an integrated version of the normal and alternative flows as presented above.

Flow of actions

1. The occupant adjusts the rooms thermostat to the desired temperature.

   If the thermostat can not be adjusted,
   then [expected behavior on user's side] the occupant of the room informs the responsible custodian about the defect.
   Exit scenario.

2. The system observes the changed value within 30 seconds (polling frequency), does a validity check and saves the value to the database.

   If the thermostat can not be polled and thus the actual room temperature can not be retrieved,
   then the system retries three more times to poll the thermostat (every 30 seconds)
   If the problem persists,
   then the system checks for communication line error and for sensor defects and prints out an error and status report for the tests that were run, logs the incident (specifying the sensor and cause for the entry) in an error log and informs the responsible custodian. Exit scenario.
   else return to 2.

   If the thermostat returns a value out of range (validity check failed),
   then the system polls the thermostat again.
   If the problem persists (validity check failed again),
   then the system informs the responsible custodian, exit scenario
   else return to 2.

3. The system compares the actual room temperature to the desired room temperature.

4. **If** actual is less than desired temperature
   then execute scenario 12 "Room temperature is too low"
   else if actual is higher than desired temperature
   then execute scenario 13 "Room temperature is too high"
   else do nothing.

Which style is to be chosen depends on preferences, circumstances and project (standards, size, user preference and environment, including tools that are used, and so on).

Further information concerning the scenario, particularly information to accommodate change management, to trace requirements and to support the development of tests, is recorded in subsequent sections of the scenario description.

Further Information

Performance and Non-Functional Requirements

Time to reach desired room temperature is 98% of all times less than half an hour.

Temperature is to be restricted to a range from +15° to +25° C.

All user-initiated temperature change requests, as well as all temperature readings shall be stored for a period of six months to allow for statistics.

Version

1.2

Owner

Pete Pattern

Type

normal

Connection to and Integration with Other Model Elements/Models

Statecharts RO2a and RO2b. Table 2a and 2b, and Figures 3 to 5 with room temperature – heating water temperature dynamics in Specification of BAS-System, Doc SE_BAS_99.312.

Open Questions

- Can heater be automatically ignited?
- Is thermostat reading accurate to 0.1°C? Accuracy of heating water temperature?

Test Planning, Test Cases

- Actual temperature < desired temp., actual temp. > desired temp., actual temp. = desired temp.
- Actual temp < down limit, act. temp > upper limit
- Communication down
- Initialization value desired temp.?
- Requirement “Temperature reached within two hours” by Marc. See his note in report 98.17.

Appendix C: Scenario Creation

In this section, we partially develop a scenario to exemplify the use of the step procedure of scenario creation as defined in SCENT. Then we transform part of the narrative scenario into a statechart to illustrate the transformation/formalization step.

The example is the well-known and familiar automated teller machine ATM. A short specification is given below.

> At an ATM the customer may inquire the balance of the account or withdraw money up to a certain amount and at given piecing (only multiples of sFr 20.- up to the personal limit may be dispensed). The customer needs a personal card and a personal identification number PIN to get access to the system and perform the mentioned banking transactions. The system interacts with a central bank system to get customer and account information and to inquire and update account balances.

1. Identify actors: In the example we identify four actors: the “Customer”, the “Service Personnel” from the manufacturer of the ATM-machine, the “Operator” of the bank or organization running the ATM and the “Bank system” holding all customer and accounts information.
2. Identify external events: Customer inserting card, entering PIN-code, choosing action, entering amount → pushing keys (numeric, control [ok, cancel, correct]), taking back card, taking cash; operator filling bills, replacing receipt rolls; service personnel opening and servicing machine.
3. System input: Cards, PINs, choices for actions, amounts, bills, receipt rolls. System output/results: Cards, balance info, receipts, cash/bills.

4. System boundaries are clear: All persons and the bank system as well as the network are environment. Customer and accounts information are kept in the bank system. Display, input panel, cash container and money dispense unit are part of the system.
5. Scenarios: (01)Inquire balance, (02)Withdraw cash, (03)Transfer funds, (04)Service ATM, (05)Reload bills, (06)Refurnish receipt roll
6. Prioritization of scenarios: First priority (01), (02), (03), (05), secondary: (04), (06) (the prioritization doesn't make much sense in the example as all scenarios are equally important in the little example chosen. So the division into the two groups just serves as an example here).

From here on we will restrict ourselves to further developing and presenting but one scenario in this example. We choose scenario (02)Withdraw cash: A customer withdrawing money at the teller machine. Each step that is described in the following paragraphs would be performed for each scenario.

7. Create a step-by-step scenario description (in the example, we do not use the full scenario template to save some space. However, the mapping to the template is straightforward):

Scenario 02: Withdraw Cash

The customer withdraws money at the ATM

Actor: Customer

Flow of actions:

1.The customer enters 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
7.The customer chooses "Withdraw cash" in the main menu
8.The system displays the cash withdrawal dialog
9.The customer enters the desired amount
10.The system returns the card
11.The system dispenses the money
12.The system prints and outputs a receipt
13.The system displays the welcome screen

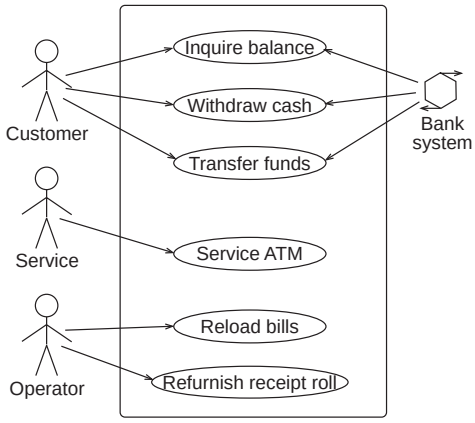
Obviously the steps are yet coarse grained (e.g. "The system checks the PIN" can be decomposed into "The system passes PIN to the bank system to be verified", "The bank system verifies the PIN" and "The bank system returns the customers authorization status"). It is also possible to indicate choices and alternatives yet (e.g. 3. **IF** the card is valid **THEN** display the "Enter PIN" dialog **ELSE** display the message "Invalid Card").

As has been done for the example scenario, the flow of actions would be broken down to a step-by-step description of actions and events at the level of single tasks in all other scenarios as well. Instance scenarios are transformed into type scenarios. Then dependency charts and an overview diagram (if desired) are created (see next step).

However, we emphasize that the process is not strictly sequential. Some scenarios might well be refined (step 10) at this point already, and abstract scenarios that are clear and distinct might be factored out at this time (step 12). In general, though, it is advisable to wait with these activities until the coarse scenarios are validated by users for the first time (step 9).

8. Create dependency charts and an overview diagram. Dependency charts are described in section 14. In appendix D an example of a dependency chart is given.

The overview diagram is a normal UML diagram depicting the actors, the scenarios and their relationships (see picture to right).



9. Scenario validation: Validation of the scenarios in a first step is done by walking customers and user through the scenarios. Later on formal user review are scheduled and conducted (Step 15). In the example we just move on.
10. Scenario refinement: The single steps in the coarse grained scenario are refined to single, in the context "atomic" actions, that is, task-level actions are broken into single working steps. The first step

    1. The customer enters the card

    does not have to be clarified as it presents a single action by the customer. The second step may well be refined:

    2. The system checks the card's validity

    There are two ways to insert refinement steps: either they do not change the established numbering scheme, breaking an existing number down to subnumbers (e.g. step 2.1, step 2.2), or the refinement steps are inserted by pushing following steps farther behind (e.g. old step 2 is refined into step 2 and step 3, old step 3 is new step 4 and so on). We prefer the second, but it is really a matter of preference. Inserting subnumbers has the advantage of not breaking references (e.g. alternatives to step 2 still reference step 2), but most often alternatives have to be refined as well (alternatives to step 2.1, alternatives to step 2.2, …). Down-shifting all the following steps has the advantage of avoiding clumsy numbering blocks.

    Step 2 in the above example can be broken into steps

    2. The system reads the card number and transfers the card number to the bank system to be validated
    3. The bank system checks the card number to be valid and returns a validation code: Code 1: Card is valid, Code 0: Card not valid, return card to customer, Code –1: Card missing or reported as stolen, withdraw card

    Step 3 in the coarse scenario description does not have to be refined, it is renumbered as step 4 and step 5 (old step 4) is refined in turn if desired or necessary:

    4. The system displays the "Enter PIN" Dialog
    5. The customer pushes a numeric key
    6. The system displays a masking character (echo key) in the input field on screen
    7. The customer pushes a numeric key
    8. …

    Control structures (selection, iteration) can be specified if helpful. Iterations are indicated by returning to a previous step, or alternatively by jumping in an alternative scenario. Normally alternatives are specified in the alternatives section of the scenario description though. Iterations may be controlled by variables or by conditions. In the example above, step 6 might better be changed to return to step 5, allowing to describe the customer entering the PIN-code in just two steps:

    6. The system displays a masking char. (echo key) in the input field on screen. Return to 5.

    Usually the PIN has a given length. Thus the iteration condition may be given by:

    6. The system displays a masking character (echo key) in the input field on screen. Return to 5 until the customer has entered six characters

    The use of variables is quite straightforward:

    4. The system displays the "Enter PIN" Dialog. Number of entered characters n=0.
    5. The customer pushes a numeric key. n = n+1.

6. The system displays a masking character (echo key). Return to 5 until n=6.

An example of a jump to an other scenario is given below:

4. The system displays the “Enter PIN” Dialog. Goto *abstract scenario 15*: ***Customer enters PIN***

An important decision to be taken before scenarios are refined is on the granularity of single scenario steps. What actions are to be considered “atomic”? What level of abstraction should be chosen? The answer heavily depends on the intended audience. Who is working with the scenarios? Are the scenarios used to capture requirements and/or domain knowledge (customer and/or domain expert has to understand them, the system is watched as a black box and system internals are not modeled), are the scenarios used as a design tool (explore and evaluate different design alternatives, developers work with the scenarios, system internals are of interest to a given extend), are the scenarios used for testing (functional test cases have to be developed, alternatives, data manipulation, equivalence classes in input data, boundary values and the like are of special interest, structure and completeness of scenarios is important)?, and so on.

11. Alternative flows: Alternatives (including exception handling) are described in our method in a separate section of the scenario description. That is, alternative flows of actions, and reactions to exceptions, are separated from the normal flow of actions. The main flow of a scenario is not cluttered by many alternatives. Thus, the people working with the scenarios are not distracted from the main course of action by many interspersed alternative paths. Furthermore, alternatives are not hid in normal behavior, but distinctly listed in a special section. Of special importance is that by listing the alternatives in a special section, the developer of a scenario is forced to consciously think about and consider alternatives and exceptions that could happen in any and every single step in the main flow of the scenario. Besides, it allows for a structured and clear presentation of alternatives and reactions to exceptions. Every step of the scenario description is screened for exceptions and alternatives in the flow of actions. Questions like: What can go wrong in the step under consideration? What alternatives does the user have? What events could happen at this point? help find and specify all alternatives.

As alternatives and exceptions are listed in the alternatives section of the scenario description, they have to be bound to the action or event in the normal flow of actions where – or when – they can occur. Each exception or alternative refers to the number of the step it relates to (see example below). As more than one exception or alternative to one scenario step can exist, the numbering scheme for exceptions and alternatives is extended: a first alternative to step 1 is numbered 1a, the steps taken in the alternative are numbered 1a.1, 1a.2 and so on, a second alternative is numbered 1b, the steps in this alternative are numbered accordingly, 1b.1, 1b.2, and so on.

After having passed through all the steps of the alternative or the exception, the scenario is further executed from the next step, that is, if an exception occurred in step 3, than exception handling is done according to 3.a, 3.a1, 3.a2, …, and then scenario execution is taken up again at step 4 of the normal flow of actions. This way we emphasize that the goal of exception handling should be to recover from errors and let the user resume at the point of interruption if possible.

If scenario execution is taken up at another point in the scenario, or in another scenario, it has to be specified explicitly. Likewise, if an alternative or exception leads to termination of the scenario, it has to be indicated explicitly (use action ‘exit scenario’ as demonstrated in examples below). In case a different scenario is called, the thread of execution normally returns to the first scenario after execution of the called scenario has ended. Thus, abstract scenarios are plugable.

In the following, some examples are given.

Normal behavior: If an exceptional or alternative path is taken in step 3 in the normal flow, then first the steps of the alternative are executed, and afterward normal flow is taken up again at step 4.

3.	event or normal action leads to exception or alternative
3a.	exception handler
3a.1	exceptional action
3a.2	exceptional action
4.	normal action

However, this implicit return to the step following the exception-causing step, may be emphasized with an explicit return to 4.

3. event or normal action leads to exception or alternative
3a. exception handler
3a.1 exceptional action
3a.2 exceptional action, return to step 4.
4. normal action

If execution should be resumed at any other place (other step in the normal flow, other scenario, exit the current scenario), then an explicit return or goto action has to be specified.

Returning to the step that first caused the exception:

...
3a.2 action, return to step 3

Returning to any other step (step preceding or following the step that first caused the exception):

3a.2 action, return to step 1 or
...
3a.2 action, return to step 15

Calling another scenario to be executed before execution of current scenario is resumed at next step:

...
3a.1 action
3a.2 Goto scenario (13)Exception handling in user interaction
3a.3 further actions

Call another scenario and start execution at a given step.Termination of current scenario:

...
3a.2 Goto scenario (13)exception handling in user interaction, step 3
3a.3 Exit scenario.

'Goto'-actions obviously lead to dependencies between scenarios. These dependencies have to be depicted in dependency charts. Dependencies that are caused by exceptions and alternative flows may be specially marked (see "Concurrency, Probable Execution Order, Execution Frequency and Alternative Flows" on page 38).

In the running example, step 1 is a user action and can hardly be influenced by the system. It reads:

1. The customer enters the card

This action may have the exception that the card can not be entered (e.g. the slot is obstructed). Even though any reaction to this exception might be out of reach for the system, it still might be worthwhile to depict the exception and thus to pinpoint the problem and initiate the search for future solutions.

The corresponding entry in the alternatives section of the scenario description could reads:

1a. The slot is obstructed
1a.1 [Expected user behavior] The customer informs a human teller or calls and informs the service department.
1a.2 If a human teller was informed: The teller informs the service department
1a.3 The service department repairs the machine. Goto scenario (4)Service ATM
1a.4 Exit scenario.

Step 2 in the running example has some exceptions as well. We mention some to further illustrate the benefits of keeping alternatives and exceptions in a separate section, as well as to illustrate the numbering scheme. Step 2 reads:

2. The system checks the card's validity

There are several exceptions that lead to alternative paths. One of them is that the card can not be read: the customer has entered an invalid card.

The alternative path is:

2a. Card can not be read
  2a.1 Display error message “Card can not be read. Card may be defect.”
  2a.2 Return card to customer. Reset. Exit scenario.

If a second exception or alternative is specified it can nicely be fit taking advantage of the numbering scheme:

2b. Card has expired
  2b.1 Display error message “Card has expired. Card will be withdrawn.”
  2b.2 Withdraw Card. Log activity. Reset. Exit scenario.

In alternative flows, further exception handling steps (even if not system related) may be described in more detail if desired and helpful. This might be appropriate if processes change because of automation. By describing further process steps in the scenario, system boundaries are questioned and thought about system-based solutions to the exception is stimulated. However, the steps have to be recognizably out of system scope, and clearly do not mark requirements.

  2b.1 Display error message “Card has expired. Card will be withdrawn.”
  2b.2 Card is withdrawn. Log activity. Reset.
  2b.3 Cards that have been withdrawn are removed once a month from the ATM and are sent to the customer department.
  2b.4 Exit scenario

In case an exception occurs in the actions that are part of an exceptional or alternative flow, the exception is included as an alternative path that is marked by keyword (**If**) and indent:

2b. Card has expired
  2b.1 Display error message “Card has expired. Card will be withdrawn.”
  2b.2 Withdraw card.
  2b.3 **If** card can not be withdrawn (malfunction of pull-in mechanism),
    then Display error message “System out of service.”, inform service department
    else Reset.
  2b.4 Log activity. Exit scenario.

Sometimes, more than one alternative flow are coupled to one exception. For example, the exception of a card that can not be read may have many causes, e.g. card manufactured by another company that is not accepted at this specific ATM, valid card with broken chip or magnetic markings, dirty card, falsification, and so on. The system might distinguish between these different causes. Thus, all the alternatives are specified in a separate alternative flow. If, for example, the system can distinguish whether a card can not be read because it is a card for other ATM-systems, because the chip on the card is broke or because the card is dirty, this would result in three different flows (if actions are different!):

2a. Card can not be read (type not accepted)
  2a.1 Display error message “No proper card for this ATM. Use Purple-card only”
  2a.2 Return card to customer. Reset. Exit scenario.

2c. Card can not be read (card defect) [same effect as exception 2a., but different cause]
  2c.1 Display error message “Card can not be read. Card may be defect.”
  2c.2 Return card to customer. Reset. Exit scenario.

2d. Card can not be read (card dirty)
  2d.1 Display error message “Card can not be read because it is dirty. Use dry and soft towel to clean card. Then try again. If the problem persists please contact the customer department.”
  2d.2 Return card to customer. Reset. Exit scenario.

Often alternatives are spanning more than one (single) step in a scenario. The archetype of this kind of alternatives is the cancel action. At most any step of a user interaction scenario the user will have the possibility to cancel the actions taken so far (entries in a dialog, changes to a file, transactions in

a database based system, and so on). The effects of a cancel action taken in any step after a committing point are always the same: rollback to the last safe state. Thus the alternative can be specified in the following manner:

7-9a. The customer presses the cancel button

7-9a.1 The system cancels the current transaction

7-9a.2 Return to 6

Alternatively decisions and exceptions can be included into the normal flow:

1. **If** the slot is not obstructed, **then** the customer enters the card **else** the customer informs the service department.
2. **If** the card can be read **then** the system reads the card number and transfers the card number to the bank system to be validated, **else** the system returns the card to the customer and displays the error message “Card can not be read”
3. …

This is appropriate when only few exceptions and alternatives are present and the main flow is not much disrupted by the selections. Else the former format (separated alternatives and exception handling) should be preferred.

A problem encountered in the case of scenarios with many exceptions and alternatives to one scenario step is, that it is not easy to see, which alternative has to be taken (e.g. is it alternative 3a., 3b., …, 3x. or 3y., 1-7a., 3-5a., 3-5b., …?) This case does not normally happen (in fact it has not happened in any project using scenarios that are known to the author); but if it does it is a clear indication, that the scenario steps have not been broken down into single steps, or to manageable units respectively. These scenarios need to be redesigned and refined.

If desired steps in the description of the normal flow of actions can be annotated by the alternatives and exceptions belonging to them.

2. The system checks the card’s validity (1-3a., 2a., 2b.)

The annotation helps to keep track of how many alternatives there are to a step in the normal flow, and which exceptions affect a given action. On the other hand, it causes much overhead and trouble in case of changes in number or sequence of steps that cause steps to be reordered or renumbered.

12. Factoring out abstract scenarios: Quite obviously certain sequences in a scenario might be reused in other scenarios. In our example the whole authentication sequence will be used in any scenario of a customer interacting with the ATM. Thus it is reasonable to factor out common sequences into abstract scenarios. Abstract scenarios are “called” in normal scenarios, executed and **return to the calling scenario** to the step following the call after execution. They may however also be ‘called’ in preconditions, that is, if an abstract scenario A is listed as being a precondition to scenario B, then A has to be successfully executed before be can even start execution. If A fails, then B can not be executed, as a precondition has not been met.

In the example the authentication procedure is factored out (not taking into account the refinements done in step 10 to keep the example short):

1. The customer enters 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

This abstract scenario, call it “Authentication”, now can be called from any other scenario. For example the “Withdraw Cash” scenario would be changed to:

1. Execute scenario “Authentication”
2. The customer chooses “Withdraw cash” in the main menu

3. The system displays the cash withdrawal dialog
4. …

Abstract scenarios are a double sided sword: On the one hand they allow to factor out common sequences of actions to be kept and maintained at a single site (place in one document, …) thus helping to keep scenarios consistent and short. On the other hand they introduce dependencies that are hard to cope with in large documents. That's why abstract scenarios have to be used sensibly and considered.

13. Non-functional requirements: They are covered and collected in a separate section in the scenario description. Non-functional requirements are written either in natural language or in a form of logical or conditional expressions like for example in the UML OCL (Object constraint language). The integration of non-functional requirements is a major research topic in which many open questions still remain. An example of some non-functional requirement for the ATM system are:

> Error messages are to be marked with the 'Attention! An error occurred' symbol.
>
> Font size has to be > 14 pt in all user dialogs.
>
> Reading and checking the card shall not take longer than 3 seconds. Validating user identification and authorization shall take at most 2 seconds.
>
> PINs (Personal identification numbers) shall be numeric and at least six digits long. Only the first eight digits shall be considered.

Non-functional requirements (NFR) often cover or affect more than one scenario. Thus it is advantageous to have a central document or repository in which all NFR are collected, and in the scenarios have links to the NFR that affect the scenarios under consideration.

14. Revision of dependency chart and overview diagram: Abstract scenarios, newly found scenarios and scenarios that have been divided or joined have to be updated in the overview diagram. The diagram and the scenario description have to be kept consistent.

15. Scenario validation: The scenarios are formally reviewed by the customer and the user. Changes in requirements, problems in understandability, misunderstandings and mistakes as well as all erroneous scenario steps, descriptions and statements are reported, the scenarios are altered, revised and updated to reflect the required changes and correct all errors. This might include a whole new iteration through steps 10 to 15 of the procedure presented here. The final version is taken as a basis for the design of the system and for the development of test cases for final test of the system.

It is quite obvious in following the above step-by-step description, that in reality the development will not be a sequential flow of the mentioned activities, but rather an iterating process with many of the steps being interleaved and interwoven. Scenarios will be structured according to the template from the very beginning, exceptions and alternate flows will be captured and recorded when they are mentioned by the user/customer, and validation steps are inserted when needed. The given procedure is but a help to remember the important steps, a help not to forget significant and substantial steps.

The example scenario as it could be formulated after the iterative process described above, having been validated by the customer, is given below in the format and structured according to the SCENT scenario template (notice that we model a type scenario, not an instance scenario [step 7]. Alternatives and exceptions again have not been specified in full). We only list the characteristic scenario information and omit the sections that capture further information (section 2 in the scenario template).

Scenario Description

Descriptive, Characteristic Scenario Information

Scenario 02: Withdraw Cash

The customer withdraws the amount of money he needs at the automated teller machine (ATM)

Description, Purpose, Goal of Scenario in Global Context

A customer who has been granted a cash card and who has been given a PIN (personal identification number) withdraws money at an ATM. By supplying the PIN, access to the system is granted. The customer has to specify the amount (s)he wants to withdraw. The system then dispenses the desired amount – provided the account covers the withdrawal and enough money bills are available at the ATM. If so desired by the customer, the system prints a receipt.

Actors

Customer, banking system

Precondition

Actor has an account, a card and a PIN

Postconditions

Success: The customer has received the money as desired, the account(s) have been updated, the ATM is ready for new customer, displaying the welcome screen

Failure: No money is dispensed, the accounts are left unchanged, the ATM is reset to ready, displaying the welcome screen

Triggering Event

The customer enters a card (‘Card sensed’-event)

Normal Flow

1. Goto scenario (07)Authentication
2. The customer chooses “Withdraw Cash” in the main menu at the console
3. The system displays the cash withdrawal dialog
4. The customer enters the desired amount and pushes the ok-button
5. The system checks the amount (amount ≤ allowance, can the desired amount in the desired piecing be dispensed [enough bills in stock]?)
6. The system notifies the bank system of the cash withdrawal (+ amount), starting a transaction
7. The bank system notifies the ATM of successful entry
8. The system returns the card
9. The customer takes the card
10. The system prints and outputs a receipt
11. The customer takes the receipt
12. The system prepares the bills and dispenses them
13. The customer takes the money
14. The system commits the transaction, logs out the customer and displays the welcome screen

Alternative Flows:

1a. Scenario (07)Authentication failed (executed not successfully, that is, the success preconditions were not met)
  1a.1 Exit scenario.

2a. The customer chooses the menu option “Inquire Balance”
  2a.1 Goto scenario (01)Inquire Balance, step 2
  2a.2 Exit scenario

2b. The customer chooses the menu option “Transfer Funds”
  2b.1 Goto scenario (03)Transfer Funds, step 2
  2b.2 Exit scenario

2c. The customer pushes the cancel-button
  2c.1 Reset system. System displays the welcome screen. Exit scenario.

4a. The customer does not enter an amount in time (“Withdraw cash”-choice by customer + 60 sec)
  4a.1 System confiscates card
  4a.2 Reset system. System displays welcome screen. Exit scenario.

4b. The customer pushes the corr-button
  4b.1 System clears amount (on screen). Return to step 4.

4c. The customer pushes the cancel-button
  4c.1 Exit cash withdrawal dialog. System displays main menu. Return to step 2.

5a. Amount entered by customer > allowance
  5a.1 System displays error message “You may only withdraw <*allowance*> per day”
  5a.2 Customer confirms message
  5a.3 Return to 3.

5b. The customer enters amount ≠ multiple of 20
  5b.1 System displays error message “Only multiples of CHF 20.- may be dispensed. Please enter new amount”
  5b.2 Customer confirms message
  5b.3 Return to 3.

5c. The customer pushes the cancel-button
  5c.1 Exit cash withdrawal dialog. System displays main menu. Return to step 2.

5d. Customer does not enter an amount before pressing the ok-button
  5d.1 System displays error message “Please enter the desired amount first”
  5d.2 Customer confirms message
  5d.3 Return to 3.
  … …

8a. The card gets stuck
  8a.1 The system informs the customer displaying message “Card stuck, please report at desk”
  8a.2 The system alerts the operator

9a. The customer does not take the card in time (card returned + 30 sec)
  9a.1 … …

In the example the cancel- exceptions have not been combined in one alternative flow. This however could easily be done if actions are the same for all cancel events. Alternative 4c. and 5c. thus could be listed as

4-5c. The customer pushes the cancel-button
  4-5c.1 Exit cash withdrawal dialog. System displays main menu. Return to step 2.

Furthermore, step 1 – calling the authentication-scenario – could have been listed as a precondition, which in this case would read

Precondition

Actor has an account, a card and a PIN, the authentication-scenario (Scenario (07)Authentication) has been performed successfully (card ok, PIN ok, system access granted, main menu is displayed)

and the triggering event would accordingly have to be altered to

Triggering event

The customer chooses “Withdraw cash” in the main menu at the console.

We are well aware that the system as specified above is not correct and complete (in the sense of being apt for implementation to be used in a real system): Some key features and capabilities of an advanced ATM-System are missing (e.g. stability and availability requirements: What happens if the banking system can not be contacted [line down]? Interaction of ATM and bank system is not specified in detail), important information is missing, requirements are not specified (e.g. is it a credit or a strict debit card system? Can information be stored and altered on the card?) and the transaction concept is simplified.

However, these limitations could easily be overcome by specifying the exception handling and alternative flows in more detail, but to keep the example simple, we do not go into more detail. The example as given is sufficient to illustrate the procedure and the result (the narrative scenario) to be created.

As can be seen in this (short!) example, a scenario can be quite extensive. Thus, scenario management is a key requirement to successfully make use of scenarios in a project. Tools support is needed to keep links and references between scenarios up-to-date and to maintain the numbering scheme of steps in normal flow and corresponding alternatives.

The following paragraph serves to illustrate what scenarios will look like in a statechart representation. In SCENT, narrative scenarios are transformed into more formal statecharts, to allow for systematic Test cases development. Statecharts created from narrative scenarios will look like this (showing the normal flow of actions only, not including preconditions, any alternative flows and exceptions yet):

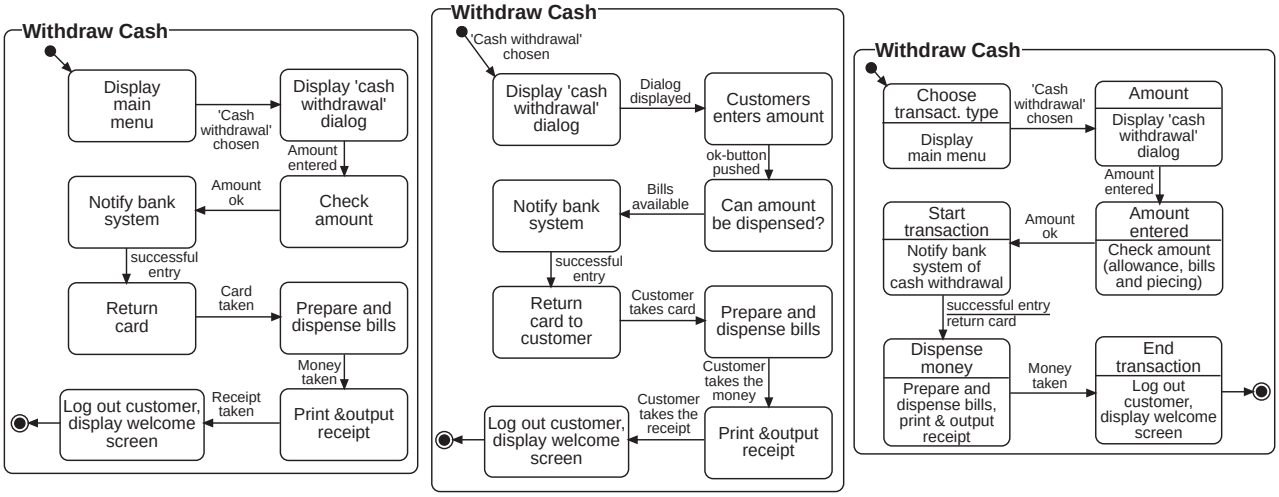


FIGURE 28. Different statecharts derived from a narrative scenario

This also illustrates, that there are many different ways to map narrative scenarios to statecharts, many of which are semantically equivalent, and just as correct, understandable and valuable as any other. However, we do suggest to use the mapping shown to the left: System actions and activities are mapped

to states (or – if appropriate – to transition actions), user actions (to be precise: actions of any actor, that is: of humans and/or other systems interacting with the system under consideration) map to events that trigger transitions. Starting states and triggering events are included in the statechart to support integration of the different scenarios/statecharts.

To take the example a step further, we include another statechart to illustrate what it looks like when alternatives and exceptions are included (leaving all the modeling, where no information has been provided in the example scenario above, yet to be done → cloudy shape in Figure 29).

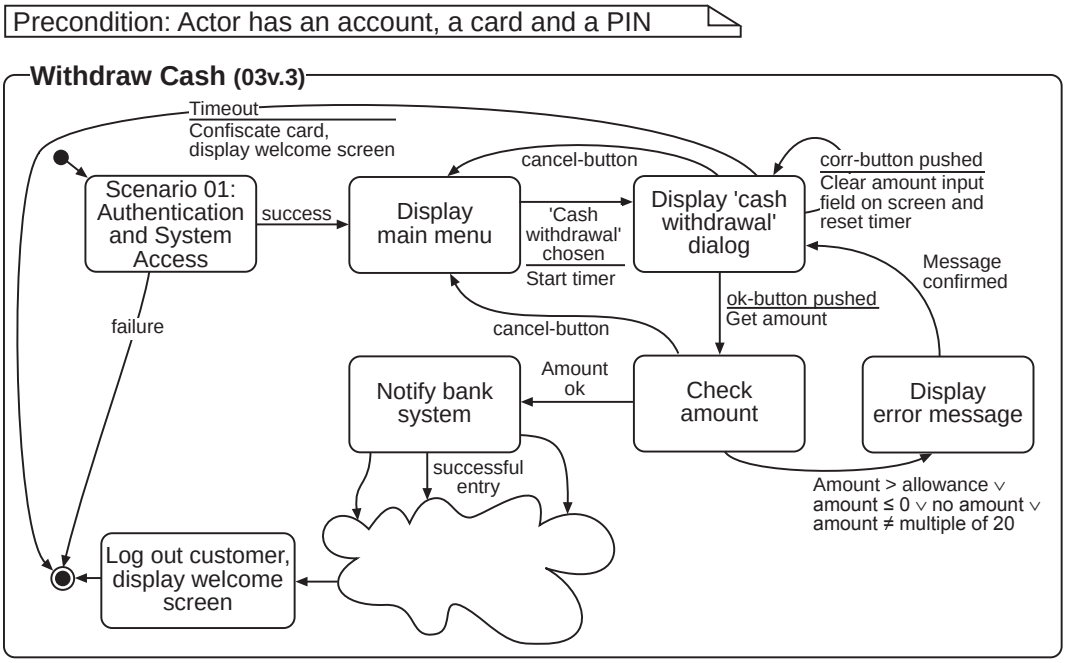


FIGURE 29. An intermediate drawing of a scenario statechart

Appendix D: Dependency Charts Example

As a short example for illustrating the creation and use of dependency charts, we choose the well-known library example [Gli95]. In a library, the user can apply for a library card to be allowed to borrow books and search for books. In the example, there are two actors: the library user (borrower of books) and the librarian. There are five scenarios in the example in which the library user is the actor: (01)*Apply for library card*, (02)*Query catalog*, (03)*Borrow books*, (04)*Return books* and (05)*Apply for deletion from user catalog*. The scenarios for the actor 'librarian' are: (11)*Register user*, (12)*Delete user*, (13)*Update user data*, (14)*Catalog book*, (15)*Remove book*, (16)*Maintain library catalog*, (17)*Query user data and status*, (18)*Query book status*, and (19)*Call overdue books*. In a real system, there certainly would be some more scenarios (and thus most probably some more dependencies…), but in the paper, to keep the example short, we limit the system to the scenarios listed above.

Some of the sequences are quite obvious (as readers are familiar with the domain and know the situation in and the context of a library very well): first a potential library user has to apply for a card, then the librarian has to register the new user, before the library may be used as many times as desired by the – now regular – user. The library user queries the catalog and borrows books up to the limit of four books per user at a time. Books have to be cataloged, before they can be borrowed. Finally, the borrowed books have to be returned in time (else an overdue note will be sent by the librarian) before they can be borrowed by another user.

The librarian on the other hand maintains the library catalog. She may query a user's status and update a user's data, once the user is registered. If a request for deletion from the user catalog is sent, the librarian will delete the user, after having called back all the borrowed books from this user. Furthermore, she will catalog new books and remove stolen, old and torn ones. A book first has to be cataloged before its status can be queried or before it can be removed.

The following picture shows the dependency chart depicting the specified system:

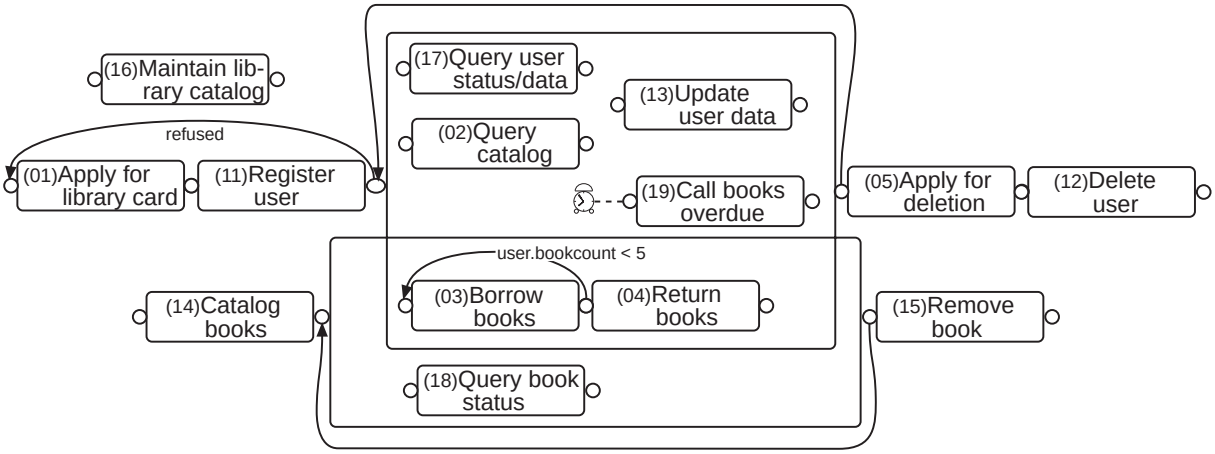


FIGURE 30. A dependency chart for the library example

Test cases are derived by traversing paths in the dependency chart, taking into account data and resource annotations and further supporting text like conditions specified. Thus, the following test cases are developed from the dependency chart shown in Figure 30. Only some test cases are specified to illustrate the procedure, the scenarios concerned are indicated by scenario identification numbers in brackets:

TABLE 5. Test sequences derived from dependency charts

Test preparation:Library catalog exists, librarian has been registered and has full access rights			
ID	*Scenarios (sequ.)*	*Expected Result*	*Description*
…	…	…	…
7.1	(01) (11) (05) (12)	User not registered, does not have user privileges anymore	User first registered, then deleted
7.2	(17)	User data and status still accessible, marked as deleted	Query deleted user's status/data
7.3	(03)	User not registered, book can't be borrowed	Borrow a book for a deleted user
8.1	(01) (11) (03)[4]	Books may be borrowed, limit reached	Register user, borrow four books
8.2	(03)	Book can not be borrowed	Borrow fifth book
…	…	…	…

Appendix E: Overview of the Dependency Chart Notation