A Practical Approach to Validating and Testing Software Systems Using Scenarios

Johannes **Ryser** Martin **Glinz**
Department of Computer Science
University of Zurich
Winterthurerstrasse 190
CH-8057 Zurich, Switzerland
{ryser, glinz}@ifi.unizh.ch

Abstract. Scenarios (Use cases) are a means to capture a system's functionality and behavior in a user-centered perspective. Thus they are used in most modern object-oriented software development methods to help elicit and document user requirements. Scenarios also form a kind of abstract level test cases for the system under development. Yet they are seldom used to derive concrete system test cases. In this paper we present a procedure to use scenarios in a defined way to systematically derive test cases for system test. This is done by formalization of natural language scenarios into statecharts, annotation of statecharts with helpful information for test case creation/generation and by path traversal in the statecharts to determine concrete test cases.

1. Introduction

In developing a software system, validation and verification are recognized as vital activities. 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 [5]. Early validation and verification thus greatly reduce error fixing and fault cost.
Testing plays an important role in validating and verifying systems. Yet test preparation and the development of test cases is often done only just before testing starts, at the end of the development process, even though analysis as well as design would greatly profit from the insight gained by developers in creating test cases and preparing tests. Moreover, testing 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 in available methods or lacking problem understanding. Any testing strategy has to address this practical issue if it is to be successfully applied. To improve testing in practice, systematic test case development and integration of test development methods with 'normal' system development methods is 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 overhead or intolerable cost.

Many strategies and approaches to testing exist. Besides established techniques like control and data flow testing or boundary analysis/domain testing [4, 17], formal languages for specification and specialized testing languages are gaining increased attention. Yet a gap is opening between the state of the art and the state of practice. The gap in-between what theoretically could be done and what really is done in practice, is mainly due to the following reasons (the list is not intended to be complete):

- **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 behind schedule and over budget already. Detecting faults causes additional delays. As a consequence, both test preparation and execution are frequently performed only superficially. Cost and

Presented at Quality Week Europe QWE'99 in Brussels. Proceedings published by the SR Institute.

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.

- **Lacking (test) documentation**: Tests are not properly prepared, no test plans are developed and tests are not documented [29].
- **Drudgery**: 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.
- **Lacking tool support**: For this reason, testing has to be supported by tools. But only limited tool support does exist. 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.
- **Formal languages /specific testing languages required**: Many test methods use formal specification languages or specific testing languages (thus requiring special training and education). Their application is extremely costly, they are difficult to apply and/or can only be applied to limited problems or very specific domains.
- **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. The quality of tests, and thus to some extent of the product, is often not assessed. Furthermore, the missing data further aggravates the problem of accurate test planning and allocation of the necessary resources.

The issues mentioned above may be addressed by various approaches. The problems of documentation and planning, for example, may be alleviated by improvements to the testing process, by use of and adherence to appropriate methods and clear definition of testing criteria, by testing strategies, or by a list of documents and deliverables that have to be produced during development of the system. Formal languages or specialized testing languages may allow for better automation of the testing process and for better tool support; 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.
The strategy last mentioned above is the one pursued in our approach: To help bridge the gap between the state of the art and the state of practice, we propose the use of scenarios, not solely for requirements elicitation and specification (as done in leading object-oriented development methods), but for system testing, too, formalizing narrative natural language scenarios into more formal statecharts and deriving test cases from statecharts. We thus allow for – and enforce (in parts) – a systematic test case development. In designing the method, we try to utilize synergies between the phases of system analysis & specification and system test.

The rest of the paper is organized as follows: Section 2 serves as an introductory chapter to define and present the problem and shortly sketch the proposed solution. In section 3 we present the basic concepts and principles of the SCENT method, and describe the individual steps in the procedure of scenario creation, formalization and test case generation. In section 4 the method presented in this paper is compared to related work and in section 5 we present some conclusions.

2. Problem Disposition and Solution Strategy

In this section we take another look at some of the problems in testing and testing methodologies. Then we present the key concepts of the SCENT method and shortly introduce the notion of scenarios and use cases.

2.1. The Problem

To deliver a software product of high quality, an efficient, reliable quality process has to be implemented and sound (engineering) principles have to be followed and adhered to. And after all that can be done to construct quality products, testing as an analytical means to software quality has to be performed in a timely and systematic manner.
But in many projects, testing is done 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. This is - at least partially - due to the following facts:

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 development process, many errors, omissions, inconsistencies and even over-specifications may be found in the analysis or design phase still. It's cheaper to remove errors in the early phases.
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. 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.
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 coverage, …). Testers are left with no definite procedure on how to derive test cases.

These concerns can be reduced by extended tool support. But as mentioned before, testing is not a simple task that can be easily automated. It is not possible at the time being to automate the whole testing process and achieve acceptable test coverage in given time for projects relying on natural language specifications [23, 26]. Therefore, proper tool support helps to alleviate the problems mentioned above. It does not, however, solve them.

2.2. A Proposal to Solve the Problem: The SCENT Approach

We propose a practice-oriented scenario-based approach to support systematic test case development, that utilizes early artifacts of the development process in later phases again, in order to realize synergies between the closely related phases of system analysis and system test. We call our approach the SCENT method - A Method for SCENario-Based Validation and Test of Software.
In SCENT, we aim at providing a method that is – or easily can be – integrated with software development methods, a method that helps developers create test cases and think about testing early on in the development process and that supports systematic generation of test cases. SCENT enables scenario-based test case development for system test of software systems, taking, as is appropriate for system test, a functional testing strategy.

The key ideas in our approach are:

1. **Use natural language scenarios** not only to elicit and document requirements, to describe a system's functionality and specify a system's behavior, but also to validate the system under development while it is being developed,

2. **Uncover ambiguities, contradictions, omissions, impreciseness and vagueness** in natural language descriptions (as scenarios in SCENT are at first) by formalizing the narrative scenarios in statecharts [10],
3. **Annotate the narrative scenarios and/or the statecharts** where needed with pre- and post-conditions, 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,
4. **Systematically derive test cases** for system test by traversing paths in the statecharts and documenting the test cases.

These key concepts need to be supported by and integrated with the development method used to develop the application or the system, respectively. Most object-oriented methods support use cases and statecharts or comparable state-transition diagrams. Thus, the basic integration of the proposed method in any one of those methodologies is quite simple and straightforward.
In section 3 we describe the method in more detail.

2.3. Scenarios

Scenarios play an important role in our approach. But even though scenarios are nowadays ubiquitous and have long been used in human-computer-interaction, strategic planning and requirements engineering, a single, formal, agreed upon definition what a scenario is, does not exist.

We define scenarios informally to be any form of description or capture of user-system interaction sequences. The terms scenario, use case and actor are defined 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 [13] – *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.*

3. Basic Principles of the SCENT- Method

By creating scenarios during requirements specification and system analysis, the requirements engineer produces a first set of abstract test cases. In SCENT, we reuse scenarios that were created during the analysis phase in test case development. The idea of using scenarios/use cases in testing is not new, however. Jacobson in his book and articles mentioned that use cases are well suited to be used as test cases for integration testing [13, 14]. Others have taken up, formalized and extended the notion of using scenarios to test a system (see for example [11] and [7]). In section 4 of this paper, a more detailed description of related work is given. But despite the ubiquity of scenario approaches, a practical method supporting testers in developing test cases from scenarios has not emerged yet [27]. The approach as presented in this paper represents ongoing research. The ideas presented are validated by application of the method in practice (see section 5).

The SCENT method comprises three main parts: Scenario creation, scenario formalization and test case derivation. All three are described in more detail in the following sections.

3.1. Scenario Creation

Many scenario processes are lacking a step procedure – a cookbook – for the creation and use of scenarios. In SCENT we define a procedure to elicit requirements and document them in scenarios. In this procedure we use a scenario template to format scenarios according to a common layout and structure. This template is not described in detail in this paper. A description of the template may be found in [24], the template itself may be downloaded from http://www.ifi.unizh.ch/groups/req/ftp/SCENT/ScenarioTemplate.pdf.

3.1.1. A Step Procedure for Scenario Creation

To create scenarios, first a list of all the persons and systems who interact with the system under consideration is created (or more precisely: a list of the roles these persons and systems play). All system in- and outputs are specified and all external events are listed. All the actors, the events and all system in- and outputs are uniquely named and a glossary of terms is created.

Having determined the actors, coarse scenarios are created capturing the main uses of the system. Ask questions like: "How does every actor interact with the system?", "How does the system react to every external event?" in order to create short natural language descriptions of system usage.

These first scenarios might be on a type or on an instance level: They may describe interaction as seen by a distinct user (e.g. Fred Brown pushes the button) or on the more abstract level of roles (e.g. Fred Brown is an operator, thus: The operator pushes the button).

Table 1: Scenario Elicitation, 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 output of the system	System output
4	Determine system boundaries	System boundaries
5	Create coarse overview scenarios (instance or type scenarios on business process or task level)	List of scenarios
6	Prioritize scenarios according to their importance and assure that the scenarios cover all system functionality	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)	Coarse grained flow of actions in scenarios
8	Create an overview diagram	Overview Diagram
9	Have users review and comment on the scenarios and diagrams	Comments and annotations to scenarios
10	Extend the scenarios by refining the description of the normal flow of actions, break down tasks to single working steps	Description normal flow of actions Hints on test case derivation
11	Model alternative flows of actions, specify exceptions and how to react on exceptions. Include hints on test case derivation	Alternative flows of actions, exception handling in scenarios
12	Factor out abstract scenarios	Abstract scenarios
13	Include performance/ non-functional rqmts./ qualities in scenarios	Scenarios, annotated with qualities
14	Revise the overview diagram	Revised overview diagram
15	Have users check and validate the scenarios (Formal reviews)	Validated scenarios

In the following steps, instance scenarios are transformed into type scenarios. The scenarios are refined by defining a step description for every scenario, and the scenarios are validated with the customer and/or the user. Alternative flows are modeled and abstract scenarios (sequences of interactions that appear in more than one scenario) are factored out.
Non-functional requirements and qualities are documented in natural language or with other appropriate means (formulas, timing constraints, pictures, graphics, screenshots, sketches, …) in a special section of the scenario description. Abstract test cases are determined and information helpful to test case development is captured in the scenario descriptions (e.g. reminders what not to forget or what specifically to test for, values of particular interest, results of activities and computations that serve as an oracle in testing, …).

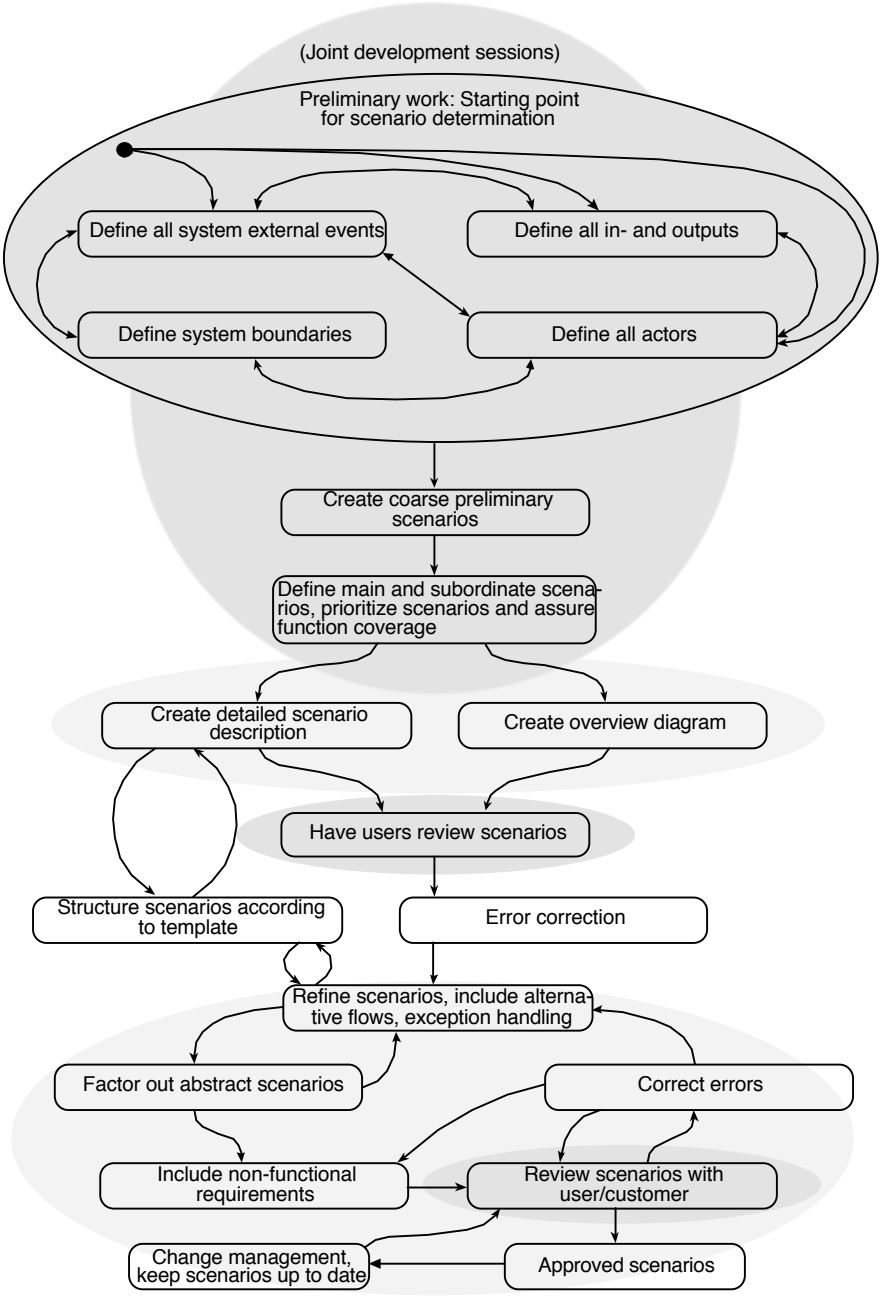

Figure 1: Scenario Elicitation, Scenario Creation and Structuring

Table 1 gives an overview of the 15 steps of the scenario creation process as defined in our method, and the purpose and results or deliverables of each step are listed.

Even though the procedure is presented as a sequence of steps, it is in reality highly iterative. Figure 1 shows the order of the activities in the scenario creation process. User involvement is depicted by shading: Darker shading indicates heavier user involvement.

3.1.2. An Example

As an example we choose the well-known and familiar automated teller machine (ATM). A short specification is given below (Figure 2).

> At an ATM the customer may inquire the balance of his/her account or withdraw money up to a certain amount and at given piecing (only multiples of CHF 20 up to the personal limit may be dispensed). The customer needs a 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. No receipts are issued.

Figure 2: The ATM machine specification

Because of space limitations only a short, partial description of the ATM example is given; the steps of the procedure are only touched upon to illustrate the procedure.
Below, excerpts of the scenario creation process for the ATM are presented. Numbers are relating to steps in the scenario creation procedure of Table 1.

1. Identify actors: In the example we identify four actors: the "*Customer*", the "*Service Personnel*", the "*Operator*" and the "*Banking system*".
2. Identify external events: *Customer inserting card*, *entering PIN-code*, *choosing action*, *entering amount*, *taking back card*, *taking cash*; *operator filling bills*; *service personnel servicing machine*.
3. Determine system input, results and output of the system. System input: *Cards*, *PINs*, *choices for actions*, *amounts*, *bills*. System output/results: *Cards*, *balance info*, *cash/bills*.
4. Determine system boundaries: All persons and the banking system belong to the environment. Customer and account information are kept in the bank system.
5. Create coarse scenarios (instance or type scenarios on business process or task level): (1)*Inquire Balance*, (2)*Withdraw cash*, (3)*Service ATM*, (4)*Reload bills*
6. Prioritization of scenarios: First priority (1), (2), (4), secondary: (3). Assure that the scenarios cover all system functionality.

We further develop only one scenario. We choose scenario (2)*Withdraw cash*: A customer withdrawing money at the teller machine.

7. Create a step-by-step scenario description:

Scenario 2: **Withdraw cash**
The customer withdraws money

Actor: Customer

Flow of actions:

1. The customer inserts the card
2. The system checks the card's validity
3. The system displays the "Enter PIN" Dialog
4. The customer enters his PIN
5. The system checks the PIN
6. The system displays the main menu
7. The customer chooses "Cash Withdrawal" from 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 displays the welcome screen

8. The overview diagram is omitted to keep the example short. It is a "standard" use case diagram in UML notation.
9. Scenario validation: Have users review and comment on the scenarios and the overview diagram. Validation of the narrative scenarios in a first step is done by walking customers and users through the scenarios. Later on formal user reviews are scheduled and conducted (Step 15).
10. Scenario refinement: The steps in the coarse-grained scenario are refined to single, in the context "atomic" actions. The first step does not have to be clarified as it presents a single action by the customer. The second step may well be refined:

2.1 The system reads the card number and transfers the card number to the bank system to be validated 2.2 The bank system checks the card number 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 3. The system displays the "Enter PIN" Dialog 4.1 The customer pushes a numeric key 4.2 The system displays a masking character (echo key) in the input field on the screen 4.3 The customer pushes a numeric key 4.4 …

11. Model alternative flows of actions, specify exceptions and how to react to them. In SCENT, exceptional flows are separated from the normal flow of actions. By doing so, the developer of a scenario is forced to consciously think about alternatives and exceptions that could happen in any and every single scenario step. Moreover the normal flow of actions thus remains uncluttered by alternatives.

    In the example above, step 1

1. The customer inserts the card

    may have the exception that the card can not be entered (e.g. the slot is obstructed). The corresponding entry in the alternatives section of the scenario description may read:

1a. The slot is obstructed 1a.1 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

12. Factoring out abstract scenarios: Certain sequences in a scenario might be reused in other scenarios. In the example, the authentication procedure is factored out:

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

13. Include non-functional requirements in scenarios: Performance requirements are included in the scenarios, qualities are appended to the scenarios. As an example, we assume that the validity-check of a card that has been inserted at the ATM must be performed in less than two seconds. Moreover, the color red is only to be used for error messages. The second step in the scenario description is correspondingly changed to read:

1. The customer inserts the card 2. The system checks the card's validity. This operation must take less than two seconds 3. The system …

    and the requirement on the use of the color red is appended to the scenario:

…
12. The system displays the welcome screen

Non-functional Requirements: The color red is to be used for error messages only

14. Revise the 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 descriptions have to be kept consistent.
15. Scenario validation: Have users check and validate the refined scenarios (Reviews). Scenarios are altered and updated according to errors and problems found (Iterate through steps 10 to 15 of the procedure).

3.2. Scenario Formalization

Scenarios are validated by users and customers throughout the scenario creation process by reviews (inspections) and walkthroughs (see Section 3.1.). Scenarios prove valuable in validating requirements: As (functional) requirements are captured in the form of interaction descriptions, the user does not have to read and validate an enumeration of required features, abstract functions and qualities that are pulled out of usage context (as they are in traditional specifications). Requirements are captured in descriptions of the flow of actions. Thus, scenarios 'naturally' bundle requirements that belong together. They do so longitudinally, that is from the start of a transaction to the end of transaction. Dependencies between requirements and the interactions between features are at least partially described in scenarios as well.

Yet natural language scenarios, as they have been created in the scenario creation process so far, suffer from the problem of all natural language specifications: Natural language is not precise, definite and unequivocal (as is shown by this very sentence: Does the negation of 'not precise' also extend to 'definite' and 'unequivocal', or is the scope of the negation limited to the adjective directly following it?). Narrative scenarios may be ambiguous, inconsistent and incomplete. Reviews by users may find some of these problems, but many inconsistencies and omissions might slip by undetected.

Formalization helps in finding and avoiding these problems. Formal languages allow for formal reasoning, (strong) verification and proof of correctness. But formal languages have their own shortcomings, too: They require knowledge of a special language, are hard to understand and their application may be error-prone.

In SCENT we take an intermediate way by converting natural language scenarios into semiformal statecharts. This formalization helps to find many omissions, ambiguities and inconsistencies, yet the graphical representation of scenarios can well be understood by users, given some guidance by the developers. Thus, the formalization is a very helpful fault-finding procedure and can be seen as a part of static testing.

3.2.1. The Formalization Step

In the formalization step, structured natural language scenarios are transformed into statecharts. This step is in SCENT informal itself. In fact, the mapping of scenarios to statecharts is a creative modeling step that can not be formalized. The statecharts created from scenarios by one developer might significantly differ from statecharts developed from the same scenarios by another developer.

We define some heuristics to support developers in the transformation step. The heuristics are:

- As soon as first scenarios have been developed (step 5 and 7 in the scenario creation procedure), create statecharts to match the scenarios.
- Create a statechart for each scenario. The normal flow, all exceptional flows and all alternatives of a given scenario are captured in one statechart.
- The statecharts are refined along with the scenarios, thus providing for a continuous validation and ongoing check for inconsistencies, omissions and ambiguities in the narrative scenarios. As the coarse overview scenarios are refined to reflect the 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.

- Model the normal flow of a scenario 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.
- Represent abstract scenarios as hierarchical statecharts.
- A single step in a scenario usually translates into a state or a transition in a statechart[1]. 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).
- External and internal events are mapped to state transitions, the triggering event is the state transition to the initial state.
- 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 [8].
- 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 will a state be entered, on what events can a state be left? 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?

Creation of scenarios and of statecharts is an iterative process. Statecharts have to be validated with the user either by inspection or review, or by paraphrasing and recounting them to the customer (end user, procurer) in a narrative style. All (important) paths are traversed; the developer guides the customer through the flows. This validation activity works hand in hand with the phase of test case derivation: The paths traversed with the customer to validate the statecharts are test cases that need to be tested in system test. Again it has to be emphasized that the process is not a sequential one, many of the activities may – at least partially – be done in parallel; they profit one from another and make use of the same artifacts.

The statecharts developed at first pass may be integrated to a full system model in a following step (as desired. It is not mandatory in the method, but might help in design as the integration of all statecharts represents a full system model).

3.2.2. Statechart Annotation

Statecharts describe the behavior of a system (how does a system behave in response to events and given conditions, ...), but let out other important information as data, performance and qualities. Nevertheless, this additional information is important for testing: Many errors are data-related and may only be found by test cases that can not be directly derived from statecharts (The sample bug

[1] 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.

statistics in [3] show that this kind of bugs might well account for up to one third of all errors). Moreover, concrete test data, that is input values and expected output, can only partially be derived from statecharts directly.

For this reason, we extend the statecharts notation to include information important for testing. In particular, the additional testing information may comprise the following:

- Preconditions (and postconditions as needed)
- Data: Input, expected output and ranges
- Nonfunctional requirements.

The information is captured in annotations as shown in Figure 4.

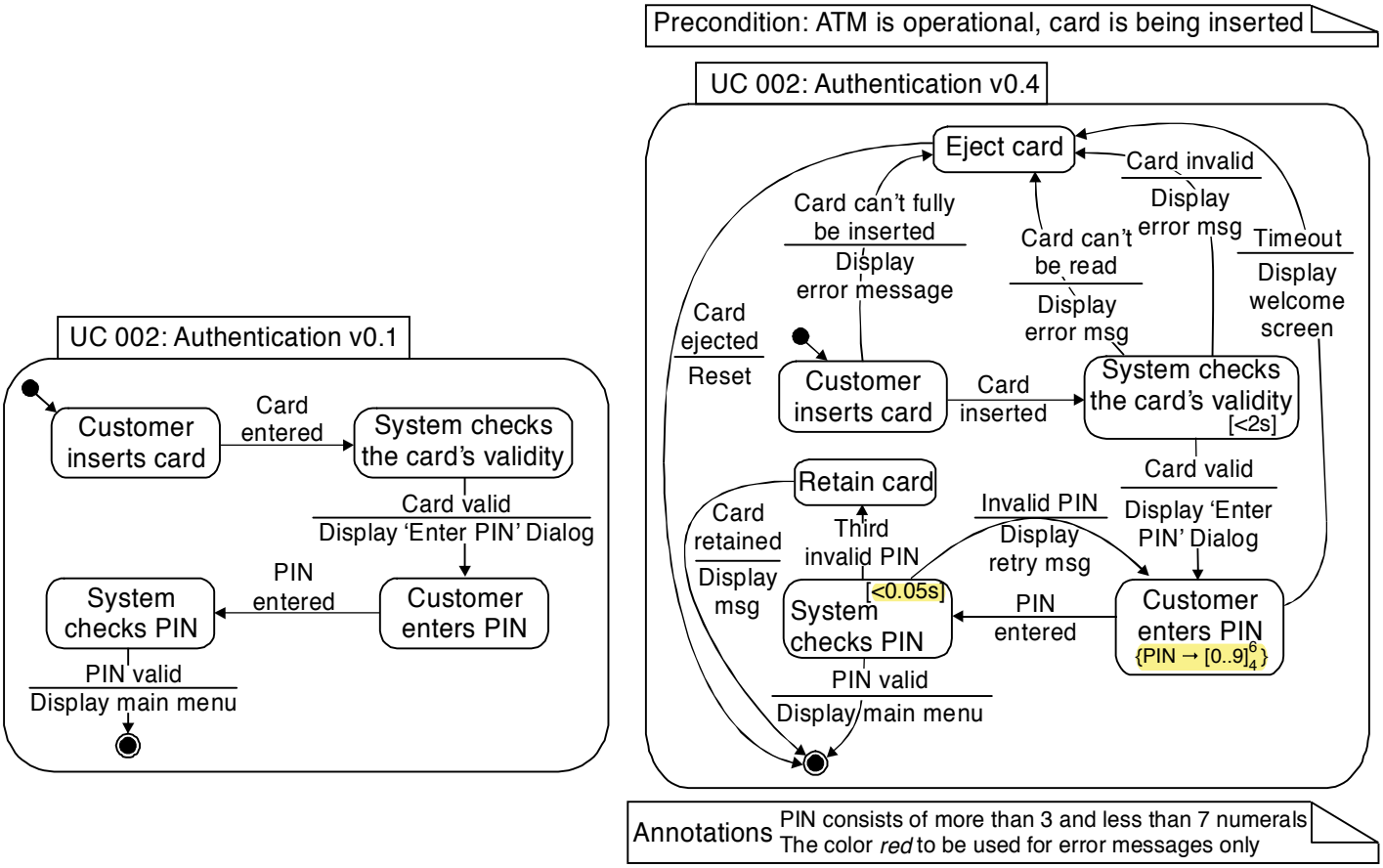

Figure 3: A statechart representing the 'Authentication' scenario

Figure 4: 'Authentication'-Statechart with alternative flows and annotations

Preconditions are captured in banner-like notes. Data is annotated to states and – if applicable – to transitions alike: Ranges are specified in square brackets (or alternatively in curly braces, if square brackets are used for other purposes – see Figure 4 for an example). Expected results can be specified for key-values. Descriptions of expected data formats and restrictions may be specified in the statechart and/or references to the specification may be inserted. If desirable (for readability or understandability of the model, for example, or if paralleled in the domain), specific states for data validation may be modeled.

Performance requirements are annotated in square brackets as well. Performance requirements spanning more than one state or transition are specified by attaching the (timing) constraint to a dashed line connected to the affected states/transitions.

3.2.3. An Example

As an example of the formalization process, we consider the abstract scenario "Authentication" of the ATM example introduced in section 3.1.2, step 12.
At first the normal flow of actions is modeled in a statechart (see Figure 3).
Then the alternative flows are modeled (see Figure 4).
Once a scenario is modeled in a statechart, the statechart is annotated as needed. Preconditions and data annotations are included in the statecharts. In the example, valid and invalid PINs are distinguished by state transitions, but no indication as to what an invalid PIN is, is made. So a data annotation may specify the ranges and the form of a PIN (see Figure 4). Furthermore, two performance constraints have been specified in the example: The verification of the card's validity shall not take more than two seconds (assuming the network connection in-between ATM and banking system to be sufficiently fast) and the validation of the PIN shall not take more then five hundredth of a second (assuming the PIN can be validated algorithmically, just knowing the PIN and an encryption key read in from the card).

3.3. Test Case Derivation

Test case derivation in the SCENT method comprises three steps:

Step 1 (mandatory). Test case derivation from statecharts.

Step 2 (mandatory). Testing dependencies among scenarios and additional tests (e.g. testing for specific qualities).

Step 3 (optional). Statechart integration and test case derivation from the integrated statechart

In this paper, only the first step is described in more detail. To support the second step, we define a new diagram type called dependency charts in SCENT. In dependency charts, timing, logical and causal dependencies between scenarios are captured and depicted. Thus, testing dependencies among scenarios is supported by test case derivation from dependency charts. For a detailed description of dependency charts see [24]. Additional test cases are developed using special testing information supplied in the scenarios: The notes taken during scenario creation and refinement now are used to enhance the initial test suite.

3.3.1. Test Case Derivation from Statecharts

In SCENT, test cases are derived by path traversal in statecharts. First, the normal flow of actions represented in the statechart is followed, then the paths representing the alternative flows of actions and the exceptions are traversed. In the method, we cover all nodes and all links in the graph, that is: all states and all transitions are covered by at least one test case. If desired, a more elaborate coverage could be chosen (e.g. switch or n-switch coverage [6, 9, 18]). Most states and many transitions are traversed more than once as annotations are used to refine test cases. Abstract scenarios are integrated in the calling scenarios, to allow for thorough tests of single scenarios. The statecharts are not integrated, though. The partial character of scenarios is thereby preserved, enabling traceability from test cases to requirements and vice versa. Furthermore, the user-oriented view of scenarios is thus promoted into testing, scenario prioritization may be utilized to determine test priorities, and finally the state-space of the (partial) solutions is thereby limited to prevent a combinatorial state/transition explosion. The incremental development procedure as encouraged by the use of scenarios is supported by not integrating the statecharts: Changes in a scenario (=changes in system usage) are usually confined to changes in one statechart and changes to test cases derived from the one statechart.

Annotations in the statecharts are taken into account in developing tests: Preconditions to 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.

The data specified in the scenarios and annotated in the statecharts help develop boundary value tests – tests that traverse the same path in the statecharts for every boundary value of data ranges as well as for a data value just above and/or below the boundary, as is done in boundary-value analysis. Domain testing techniques and data flow testing can be applied to derive further test cases ([3, 4, 17]). Furthermore, as path traversal in statecharts will only generate tests for valid sequences of events, the tester has to ensure inclusion of invalid event sequences in the test. These tests are constructed by evoking or generating events while the machine is in a state where it should not respond to the generated events. Thus, exception testing can be improved and systematized and a better test coverage can be achieved.
Notes on performance and nonfunctional requirements in the scenarios help to formulate test cases to test these properties.

3.3.2. Test Case Derivation in the Example

To illustrate test case derivation from statecharts, we present some test cases as created by path traversal of the statechart depicted in Figure 4. The first test case follows the normal flow of actions: The card can be inserted and the card as well as the PIN are valid. The next test case considers the exception of an incorrect PIN entered. Next an invalid PIN is entered (PIN too short; this test case takes into account the data annotations specified in the statechart). Finally, a third invalid PIN is entered to provoke another validation failure and traverse the *Third invalid PIN* link (see Table 2).

In the statechart, the paths that have been traversed are marked. If the developer/tester encounters a data annotation, be it on a link or in a state, he/she creates a test case for every boundary value/one above/one below. This means that in the example, the tester has to develop a test case using a key that is too short (three numerals), a key that is four and six numerals long, respectively, and a key that is too long (seven numerals). If a character key was used, the tester would test for keys using inadmissible characters. In this example it is obvious that state-transition tests are not sufficient: Even though the state-machine will differentiate between valid and invalid PINs, it does not indicate why a key is invalid. Is it because the PIN does not meet required syntactical or formal attributes (length, only admissible characters), or is it because the user has entered a PIN that is syntactically correct but not valid as it is not the user's PIN? And even if this difference is modeled by distinct transitions (incorrect PIN vs. invalid PIN), still the tester has to test for all kinds of (syntactically) incorrect entries, e.g. key to long, key to short, key has inadmissible character, key does not include at least one non-alphanumerical character, and so on.
Annotations of performance requirements are tested for in like manner.

Table 2: Test Cases for the ATM Example

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

The test cases in Table 2 might well be refined to reflect more details in requirements: The normal flow of actions captured in test case 1.1 in Table 2 above can be broken down to single steps as illustrated in Table 3. The decision on the level of abstraction in testing (to what detail shall be tested) depends on the kind of test (unit level, integration, system) to be run and on the testing that has been performed before (full tests on a system level can not be done because of the resulting test suite size or at least is not economical to do so).

Table 3: Refined Test Cases for the ATM Example

Test preparation:		ATM operational, card and PIN (1234) have been issued, card is being inserted	
ID	*State*	*Input/Actions/ Conditions*	*Expected output*
1.1	Card sensed	Card is taken in	Card inserted, validation screen displayed
1.2	Card inserted	Validate Card	Card is valid, 'Enter PIN'-dialog displayed
1.3	Card valid	Customer enters PIN	PIN (1234) entered in time, validation screen displayed
1.4	PIN entered	Validate PIN	PIN is valid, main menu displayed
…	…	…	…

4. 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 also in the UML (Unified Modeling Language), and many different approaches have been developed over the last couple of years. Yet, in the area of testing, only few scenario-based approaches exist. In the following, we review some approaches with regard to use case creation, scenario formalization and support for testing activities:

- Jacobson's Use Case Approach [13-15] 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 us scenarios in testing. Scenarios are not formalized in the approach. No specific description format or template is advocated. Furthermore, use cases – 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/may not be tested for.
- Hsia et al. [11] 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:
  - 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.

- 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 and requires special training and skills on the side of the developers. Furthermore 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.
- 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.
- In our approach, statecharts are annotated with preconditions, data and nonfunctional requirements to enhance the creation of test cases. No equivalent concept has been defined in the Hsia approach.
- Dependencies and relationships among scenarios may in SCENT be modeled in dependency charts [24] and dependencies may be tested for accordingly. In their approach, Hsia et al. do not propose any way to handle inter-scenario dependencies.

- Firesmith [7] extends the scenario approach to model scenario lifecycles and the relationship between different scenarios. The benefit of scenarios in testing is only hinted at, the development of test cases is not addressed at all.
- Regnell et al. [20, 21] 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 [20] is to present improvements to the OOSE/Use Case Driven Analysis UCDA approach of Jacobson [13] by identifying weaknesses and problems in UCDA and defining a possible solution. [21] focuses on the representation, extending the former approach to include a hierarchical structured representation. Testing and the derivation of test cases is not an issue in the approach.
- Potts et al. [19] 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 reasoning and discussion about requirements. In their approach, 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 work.
- Lee et al. [16] use Petri nets for the analysis and integration of use cases. They emphasize the importance of incremental specification of partial system behavior and of consistency and completeness 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 some of the approaches mentioned above, this approach tries to integrate and analyze use cases. However, it does not define a procedure for test case derivation from scenarios. No scenario creation procedure is specified either; scenario descriptions are input to the method.
- Message sequence charts MSCs were used in different approaches to formalize scenarios or to capture requirements of systems (see [1] for an example). 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 complex system descriptions.

5. Conclusions

In this paper we have presented the SCENT method, a scenario-based approach to support the tester of a software system in systematically developing test cases. This is done by eliciting and documenting requirements in natural language scenarios, using a template to structure the scenarios. Scenarios then are formalized into statecharts. Finally, test cases are derived by path traversal in statecharts.

The method introduced in this paper has been applied in practice to two projects at the ABB Research Center in Baden/Switzerland. First experiences are quite promising as the main goal of the method, namely to supply test developers with a practical and systematical way to derive a first set of test cases, has been reached. The projects in which the method was applied were applications to remote monitoring of embedded systems [12].

The use of scenarios was perceived by the developers as helpful and valuable in modeling user interaction with the system. Developers especially appreciated the integration of the users and the direct feedback they received in creating and refining scenarios.

Not surprising, the creation of coarse overview scenarios and the iterative refinement of scenarios down to sequences of atomic actions proved to be valuable and was very much appreciated by users and developers.

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 methods were:

- Contrary to the positive experience in modeling user interaction in scenarios, it was troublesome to model system internal interaction and the interaction with other systems in scenarios as well. Instead of using scenarios, the developers preferred to create state machines and/or other models directly.
- Developers did not want to specify scenarios down to the level needed for test case design. Scenarios were used to model user interaction with the system at an abstract level, then refined to cover all the systems tasks. At this point it was not easy to convince the developers that further refinement and specification was needed, if scenarios were to be used for test case generation.
- Scenario management was considered a major problem throughout the development. As the primary scenarios created in SCENT are natural language descriptions and as they 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 (e.g. all the links between scenarios and models derived from scenarios like class and behavioral models, and more especially the links between scenarios and other parts of the specification), and finally keep scenarios up-to-date when requirements, the environment and the developers' understanding of the problem are changing.

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

Scenario formalization is not free, some extra work is needed; in fact, the development of statecharts might take quite some time and bind some resources. However, 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 systematical way to develop test cases, test coverage is improved.

The cost of developing the statecharts is justified by the benefits of an improved specification and enhanced testing.
Test case creation on the other hand was unproblematic (as expected).
Future work will be done in the direction of defining a coverage criterion for requirements captured in scenarios or in other natural language documents and descriptions, and measuring/quantifying the improvement in coverage gained by applying the SCENT method. We are also aiming at more tightly integrating non-functional requirements into scenarios and statecharts.

References

[1] M. Andersson, J. Bergstrand, "Formalizing Use Cases with Message Sequence Charts," in *Lund Institute of Technology*: Lund, 1995.

[2] 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.

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

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

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

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

[7] 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.

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

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

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

[11] 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.

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

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

[14] I. Jacobson, "Basic Use Case Modeling," *Report on Object Analysis and Design*, vol. 1, # 2, pp. 15-19, 1994.
[15] I. Jacobson, "Basic Use Case Modeling (cont.)," *Report on Object Analysis and Design*, vol. 1, # 3, pp. 7-9, 1994.
[16] 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.
[17] G. J. Myers, *The Art of Software Testing*. New York: John Wiley & Sons, 1979.
[18] 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.
[19] C. Potts, K. Takahashi, A.I. Anton, "Inquiry-based Requirements Analysis," *IEEE Software*, vol. 11, # 2, pp. 21-32, 1994.
[20] B. Regnell, K. Kimbler, A. Wesslén, "Improving the Use Case Driven Approach to Requirements Engineering," *Proceedings 2nd International Symposium on Requirements Engineering*, York, England, 1995.
[21] 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.
[22] C. Rolland, C. Souveyet, C. Ben Achour, "Guiding Goal Modeling Using Scenarios," *IEEE Transactions on Software Engineering*, vol. 24, # 12, pp. 1055-1071, 1998.
[23] J. Ryser, S. Berner, M. Glinz, "On the State of the Art in Requirements-based Validation and Test of Software," Universität Zürich, Institut für Informatik, Zürich, *Berichte des Instituts für Informatik* 98.12, Nov 1998.
[24] J. Ryser, "SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test," to appear as a technical report at University of Zurich, Institut für Informatik, Zürich, 1999.
see: www.ifi.unizh.ch/groups/req/ftp/SCENT/SCENT_Method.pdf
[25] S. Somé, R. Dssouli, J. Vaucher, "Toward an Automation of Requirements Engineering using Scenarios," *Journal of Computing and Information*, vol. Special issue: ICCI'96, # 8th International Conference of Computing and Information, pp. 1110-1132, 1996.
[26] I. Spence, C. Meudec, "Generation of Software Tests from Specifications," *SQM'94 Second Conference on Software Quality Management*, Edinburgh, Scotland, UK, 1994.
[27] A. G. Sutcliffe, N.A.M. Maiden, S. Minocha, D. Manuel, "Supporting Scenario-Based Requirements Engineering," *IEEE Transactions on Software Engineering*, vol. 24, # 12, pp. 1072-1088, 1998.
[28] K. Weidenhaupt, K. Pohl, M. Jarke, P. Haumer, "Scenarios in System Development: Current Practice," *IEEE Software*, vol. 15, # 2, pp. 34-45, 1998.
[29] T. Yamaura, "How to Design Practical Test Cases," *IEEE Software*, vol. 15, # 6, pp. 30-36, 1998.