Using Dependency Charts to Improve Scenario-Based Testing

Management of Inter-Scenario Relationships: Depicting and Managing Dependencies between Scenarios

Johannes Ryser
University of Zurich, CH-8057 Zurich and
ABB Corporate Research
CH-5405 Baden-Daettwil
+41-56-486 83 08 / +41-1-63 54572
johannes.ryser@ch.abb.com
ryser@ifi.unizh.ch

Martin Glinz
Institut für Informatik
University of Zurich
Winterthurerstrasse 190
CH-8057 Zurich, Switzerland
+41-1-63 54570
glinz@ifi.unizh.ch

ABSTRACT

Scenarios (use cases) are used in many modern software engineering methods for capturing requirements and specifying a system. Yet prominent and renowned approaches like the UML (Unified Modeling Language [1]) are missing a concept for modeling dependencies and relations between scenarios and offer only little support for the management and description of scenarios and inter-scenario relationships. Furthermore, analysis scenarios are hardly ever used in testing, even though scenarios form a kind of abstract test cases.

In this paper we introduce a new kind of chart and a notation to model dependencies between scenarios. We call it dependency chart. We introduce a method to use scenarios and dependency charts in testing to support testers to systematically develop test cases for system test.

Keywords

Scenario, use case, scenario-based testing, dependencies between scenarios, scenario management

1 INTRODUCTION

Validation and verification (V&V) are important activities in developing a software system. According to the IEEE Standard Glossary of Software Engineering Terminology, validation and verification is "*the process of determining whether the requirements for a system or component are complete and correct, the products of each development phase fulfill the requirements or conditions imposed by the previous phase, and the final system or component complies with specified requirements*" [7].

Consequently, requirements validation is the activity of ensuring that the requirements adequately and completely state the needs of the user and procurer, and verification comprises all activities that help check that the results – intermediate or end results – conform to their specification. Testing, being one of several possible V&V activities, aims at finding errors/defects and establishing confidence that the implemented system conforms to the requirements.
Despite the existence of alternative V&V methods, as for example inspections or formal verification, most software developing organizations still use testing as their sole V&V activity.

Yet, notwithstanding that testing is the principal V&V activity, testing often is done in an unsystematic, unstructured way. Tests are hardly planned for, they are not documented, and testing is not started with until implementation is almost done.

Presented at the 17th International Conference on Testing Computer Software TCS'2000 in Washington, D.C.

Validation and verification has its problems as well: if a formal language is used in specifying the system, properties of the system may be checked automatically, the model may be checked for correctness, internal completeness and consistency. But reading and understanding the model requires special training and education; writing specifications is hard and error-prone work. Validating the specification with users is often not possible due to lack of understanding. If natural language techniques are used, the benefits and problems just reverse. The specification is now understandable to almost anybody, and it is by far easier to write natural language specifications. However, natural language specifications are hard to check for omissions. Moreover, natural language is ambiguous and vague, if not special care is taken to avoid it.

It would be desirable to have a language that is as easy to use as natural language and as rigid, consistent and concise as a formal language. In using scenarios (use cases), we try to walk the line in-between those two: on one hand, we use natural language to specify the system, on the other hand, we formalize scenarios and thereby try to alleviate some of the problems of using natural language.

Scenarios are – in the context of requirements and software engineering – sequences of interactions between users and a system. As such, they are mainly used in the analysis phase to elicit, capture and document requirements. However, scenarios are helpful in many other activities as well: manuals need to be written, taking a user-centered view – as do scenarios; requirements, design and implementation need to be validated by users (again a user-centered view of the system is needed); and, as test cases need to be developed, scenarios come in handy again, as they are abstract level test cases and may well be (re-)used in testing.

However, current scenario approaches do not exploit many of the potential benefits of scenarios. Most of them use natural language for writing scenarios, thus inheriting the problems of natural language (vagueness, ambiguity, etc.). Dependencies among scenarios are hardly ever considered and modeled, and non-functional requirements are seldom included in scenario models in a systematic way. Furthermore, most approaches use scenarios only for capturing and describing requirements. They do not reuse them for V&V activites.

In the SCENT-Approach, we take up some of the problems of current scenario approaches and propose a procedure to more fully exploit the advantages of scenarios in validation and testing. Moreover, we define a new diagram type which is used to capture and depict dependencies between scenarios. The enhanced method enables the tester to take into account the dependencies and relationships between scenarios in testing and thus supports improved test case development.

The rest of this paper is structured as follows: scenarios are discussed in some detail in section 2, followed by an overview of the SCENT-Method. In section 3, the basic principles of the SCENT-Method are presented. In section 4, we presents dependency charts as a means to depict and manage dependencies among scenarios. We conclude the paper by presenting some experiences made in applying the approach in two industrial projects in practice and we discuss the conclusions drawn from these results.

2 METHOD OVERVIEW

In this section we introduce some key concepts of scenarios and then present an overview of the SCENT-Approach [13, 14]. The details are covered in section 3.

2.1 Scenarios

In the context of this paper, scenarios are defined as descriptions of (possible, future) sequences of interactions between a user and an (existing or imagined) system (see definition below).

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	- A sequence of interactions between an *actor* (or *actors*) and a system triggered by a specific actor, which produces a result for an *actor* [9]. A type scenario in our terminology.
Actor	- A role played by a user or an external system interacting with the system to be specified

The use of scenarios as a means for eliciting, documenting and validating requirements has gained much attention and spread over the last couple of years. In particular, the work of Jacobson [9], who coined the term "use case" (see definition above) and the success of UML have made the notion of scenarios popular.

Scenarios have been incorporated in most object-oriented software engineering methods. Yet, in most of these methods, scenarios are defined only informally. Moreover, dependencies between scenarios are not modeled in most methods. Jacobson excludes concurrency and dependency modeling in use case diagrams, save for «extends» and «uses» associations [10, 11]. The rationale behind this is to keep use case models simple ("Thus, actors and use cases are the only occurrences, phenomena, or objects in a use-case model – no more, no less" [10]). Timing, data, resource and causal dependencies are not to be included in the use-case model. Other dependencies as for example structural dependencies (use case aggregation) are not encouraged because they "support functional decomposition, which would lead easily to a functional rather than object-oriented structure" [11].

However, scenarios are partial descriptions of system behavior, and as such, they are often applicable to restricted situations only. In most applications, the ordering of scenarios is at least partially not arbitrary and copious dependencies between scenarios exist. In our opinion, dependencies between scenarios carry important information about a system and hence have to be modeled, in particular if scenarios are to be used for testing.

2.2 The SCENT-Approach

Testing plays an important role in validating and verifying (software) systems, thus it should 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.

To alleviate some of the problems in testing, we propose a scenario-based approach to support systematic test case development. We aim at improving and economizing test case development by three measures.

- We (re)use and utilize 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).
- We integrate 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.
- We define a method how to develop test cases systematically.

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

The key ideas in our approach are:

1. Use natural language scenarios not only to elicit and document requirements and to specify the behavior and functionality of a system, but also to validate the system while it is being developed,
2. Uncover ambiguities, contradictions, omissions, impreciseness and vagueness in natural language descriptions (as scenarios in SCENT are at first) by formalizing narrative scenarios in statecharts [6],
3. Annotate the statecharts - where needed and helpful - 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 derived from narrative scenarios, and in dependency charts that depict the interrelations among scenarios, choosing a testing strategy as appropriate and documenting the test cases.

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

methodologies is quite simple and straightforward. In section 3 we describe the method and integration in more detail.

3 BASIC SCENT-METHOD

At first, scenarios have to be created. Scenarios in the SCENT-Method are created by a well-defined procedure. In this paper, we only present a short summary of the procedure, more details may be found in [13], where also an illustrating example is given. The method is fully described in a technical report [14].

Scenario creation is an iterative process, involving the customer (the procurer and the user), to specify the behavior, the functional requirements as well as the qualities of the system to be developed. In the following sections we describe the scenario creation procedure (section 3.1), scenario validation and formalization (section 3.2) and the annotation approach to capture information important for testing (section 3.3). Test case derivation from statecharts is described in section 3.4. The description of these topics is kept short to leave more space for the key theme of the paper: the notion of dependency charts and their use in testing (section 4).

3.1 Scenario Creation

Most scenario processes used today are lacking a step procedure for the creation and use of scenarios. For this reason we define a procedure for eliciting requirements and documenting them in scenarios (see Table 1). We use a scenario description template to document and format narrative scenarios. Thus we enforce adherence to a common layout and structure. Furthermore, we incite and encourage the developer to record information important for testing while in the analysis phase yet. For this purpose, the template features a section where test planning has to be done and test cases have to be specified. Diagrams, non-functional requirements and qualities are included in the scenario description as are pre- and postconditions for every scenario.

The procedure is straightforward and easy to apply. The individual steps are listed in Table 1. It has to be emphasized though, that the procedure is highly iterative, parallel and intertwined, that is: the steps don't have to be performed in a sequence, rather the list is to be used as a checklist and as a general outline directing the developers in the process of scenario creation. In [13], an illustration of the iterative procedure and the intertwining of individual steps is given.

Table 1: The scenario creation procedure

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

Some short notes concerning the procedure are appropriate: actors, in- and outputs and events (as determined in steps 1-3) are uniquely named. A glossary of terms including a description of all actors, in- and outputs and all events is created.

The coarse scenarios created in step 5 are short natural language descriptions of the interaction and do not feature a step-by-step description yet. In step 6, scenarios are prioritized, thus allowing for release planning.

Validation activities are interspersed throughout the development process (see step 9 and 15).

3.2 Scenario Formalization and Validation

Validation of scenarios is an integral part of the SCENT-Method. As scenarios are but narrative natural language descriptions at first, validation and verification activities are very important, because natural language is inherently ambiguous, it allows for vagueness and gives rise to inconsistencies. Natural language descriptions can not be checked – neither formally nor automatically – for completeness and absence of the properties mentioned above. Furthermore, the meaning of language sentences is always defined in a context and builds on implicit and environmental, contextual knowledge; it is interpreted on common background of speaker and listener. Specifications however should be unambiguous and consistent, exempt of vagueness and interpretability and should state the needs of the users adequately and completely.

In SCENT, validation and verification is supported in two ways:

- Walk-through and reviews are integrated in the scenario creation procedure (see steps 9 & 15 in Table 1)
- Narrative scenarios are formalized in statecharts.

Validation can only be done by the user. Inspections are an appropriate means to validate the narrative scenarios. Validation of narrative scenarios by users is accompanied by verification steps by the developer. Verification is supported in the SCENT-Method by formalization: by converting natural language scenarios into statecharts, many omissions, ambiguities and inconsistencies can be found.

The formalization step is the transformation of structured natural language scenarios into a statechart representation. Contrary to other approaches ([3, 15]) 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 (see [14]).

Creation of statecharts from narrative scenarios is a creative step. A statechart developed by one developer may differ significantly from a statechart developed by another developer from the same narrative scenario (as a model developed from the same specification by different developers may differ). This is not problematic, but emphasizes the need for further validation: the statecharts developed from narrative scenarios have to be validated with users either by inspection or review, or by paraphrasing sequences of actions in the statecharts in a narrational style. All (important) paths are traversed; the developer guides the customer through the flows. This validation activity works hand in hand with the phase of test case derivation: the paths traversed with the customer to validate the statecharts are test cases that need to be tested in system test.

3.3 Statechart Annotation

The main problem in using narrative scenarios and derived statecharts as a means to develop test cases is that they usually are on a level of abstraction that does not allow derivation of concrete test data – that is of input values and expected output – directly.

For this reason we extend the statechart concept to include information important for testing and test case derivation. In particular, the additional testing information included in every statechart should comprise the following: preconditions, data (input, output, ranges) and nonfunctional requirements. The information is captured in annotations: preconditions are annotated in banners or as conditions to the initial transition in a statechart, data is recorded in states or on transitions (e.g. in the action) in rectangular brackets, using any appropriate form (boundary values, lists, sets, placeholder examples, BNF, …) to describe the data format and ranges. Performance requirements are annotated to states (or if needed even to actions or transitions) in rec-

tangular brackets. Qualities are captured in notes-boxes attached to the statecharts. The details of the annotation are described in [14].

An example of a narrative scenario and of statecharts, developed from narrative scenarios and annotated with needed information, may be found in [13].

3.4 Test Case Derivation from Statechart

After narrative scenarios have been transformed into statecharts, test cases for system test are generated by path traversal in the statecharts. Any method to traverse all paths in finite state machines according to a given criterion can be used to derive test cases. The method of our choice is simply to cover all links, thus reaching a coverage for the state graph comparable to branch coverage C2 in structural testing. A more elaborate coverage could be chosen as desired (e.g. switch or n-switch coverage [2, 5, 12] 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.

4 DEPENDENCY CHARTS

Scenarios are partial descriptions of system behavior that are applicable in restricted situations only. In most systems and applications, the ordering of scenarios is at least in parts not arbitrary. Dependencies between scenarios do exist. As has been mentioned before, most of the current scenario approaches do not facilitate a (graphical or defined textual) representation of dependencies between scenarios. For testing purposes however, such a description of dependencies is not only helpful, but needed. If scenarios are to be used to derive test cases, then the dependencies between scenarios have to be tested also. A diagram that shows the dependencies between scenarios is not only useful in testing, it is of equal importance in design and implementation.

In the following section, we therefore motivate the introduction of a new diagram type which we call dependency chart. The concepts and notation used in dependency charts are introduced and their use in testing is outlined. A short example is given to illustrate the notation and use of dependency charts.

4.1 Motivation

The motivation to introduce yet another diagram stems from the sore need for a means to depict the dependencies and relationships among scenarios which arises notably in testing. This need for a notation to capture interrelations between scenarios does arise when creating and using scenarios to capture requirements and specify the system, too. But in these phases it is not very demanding, as scenarios are not the only system model, but might be accompanied by class, object and other models. Furthermore, dependencies are captured in pre- and postconditions. The developer will build a mental model of the relations and dependencies between scenarios if a defined procedure for depicting and handling the dependencies is missing. He/she might use sketches of any kind to discuss these matters. In testing however the need for a defined model of dependencies and relations among scenarios arises more prominently: it is not possible to test the system thoroughly if the dependencies and relations are not considered appropriately. Furthermore, it is not possible to make tests repeatable and traceable if dependencies are not documented, thus making test case development comprehensible and repeatable.

For this reason we introduce a new kind of diagram, which is used to depict the dependencies and relations among scenarios. We call this diagram *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.

4.2 The Concepts

Scenarios are dependent on other scenarios in many ways: one scenario needs to be preceded by another one, a scenario might not be run in parallel with another scenario, data needed by scenario A is prepared by scenario B in a producer-consumer-relation, scenario A calls scenario B in its normal flow or to handle an alternative or exception, and so on…

Dependencies between scenarios fall into one of the following categories: abstraction, temporal and causal dependencies. Abstraction dependencies are introduced into the model by hierarchical decomposition of model elements, by aggregation, generalization and refinement structures. Temporal dependencies establish a sequence dependency between scenarios, e.g. a scenario needs to be preceded or followed by another scenario. Temporal dependencies map to strict sequences (see Figure 1) or to real-time dependencies in dependency charts. 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 (Fig. 1): scenario B may be executed any time the conditions hold true. Data (e.g. specific data items need to be created before scenario B is executed) and resource (e.g. an electronic connection needs to be established before scenario B can be performed) dependencies are special cases of causal dependencies.

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.

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.

4.3 The Notation

To depict dependencies between scenarios, an existing notation, e.g. the scenario overview diagram of Jacobson et al. [9], might have been chosen and extended appropriately. However, we think it is advisable to use a distinct notation to emphasize the difference of the new diagram from existing ones in meaning and use.

In dependency charts, scenarios are shown as rectangles with rounded corners and circular connectors. Scenarios without any connecting lines (dependency lines) may be executed as many times as desired and in free order, even in parallel if appropriate.
Sequences are shown by connecting scenarios by their circular connectors, thus indicating a strict sequence (the second scenario must follow the first, the first must be followed by the second – used for temporal dependencies. As strict sequences are connecting two nodes, the connectors represent the edge) or by connecting two scenarios with a dependency line – this is called a loose sequence (if the second scenario is to be executed, it must be preceded by the first – used for causal dependencies).
Alternatives are often implicitly specified (usually by scenario name), as any unrestricted scenario may be executed alternatively to other unrestricted scenarios. If alternatives have to be shown explicitly, they are shown by rectangular connecting lines.
Iterations are depicted on dependency charts by backward sloping arrows connecting the appropriate scenario connectors. Accidental concurrency (two scenarios may run in parallel) is implied by using unrestricted scenarios. If concurrency has to be enforced or is to be prevented, the scenarios are connected using the appropriate symbols (Fig. 1). Real-time dependencies are indicated by the alarm clock symbol. Abstract scenarios (as factored out in step 12 of the scenario creation procedure) are depicted as foldouts. To structure the dependency model we use a hierarchical structure. Scenarios which belong together according to some criterion may be packaged in a box (Fig. 1). 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.

A small example is given below to illustrate the representation of scenarios and their relations and dependencies in a dependency chart (Figure 2).

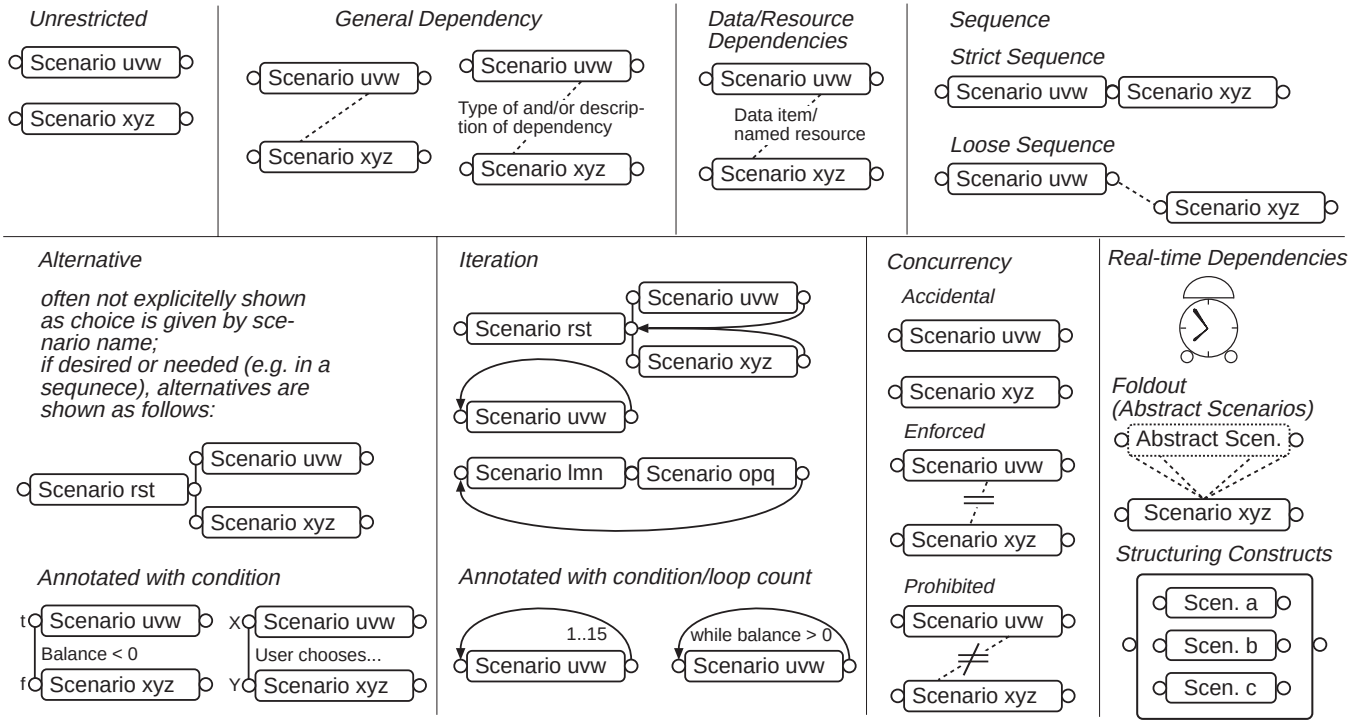

Figure 1: The notation for dependency charts

A further discussion of dependency charts may be found in [14].

4.4 Test Case Derivation from Dependency Charts

Dependency charts are particularly 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).

4.5 An Example

As an example illustrating the creation and use of dependency charts, we choose the well-known library example [4]. 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:

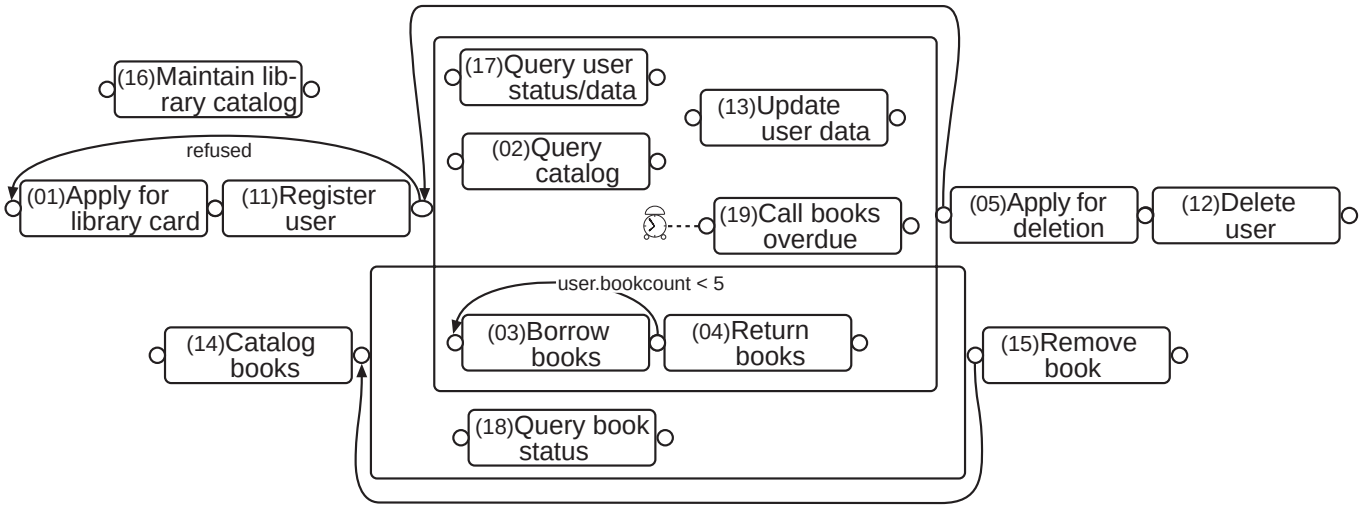

Figure 2: A dependency chart for the library example

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 2. Only some test cases are specified to illustrate the procedure, the scenarios concerned are indicated by scenario identification numbers in brackets:

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
…	…	…	…

5 CONCLUSIONS

In this paper we have presented the SCENT-Method, introducing a diagram/chart to capture dependencies between scenarios and shortly sketching the use of this dependency graph to create test cases.
The SCENT-Method has been applied in practice to two projects at ABB in Baden/Switzerland. First experiences are quite promising as the main goal of the method, namely to supply test developers with a practical and systematical way to derive test cases, has been reached. An experience report may be found in [14]. The projects in which the method was applied were applications to remote monitoring of embedded systems [8].

The use of formalized scenarios as a means to support systematic test case development proved helpful in improving testing. As single scenarios are transformed into statecharts from which test cases are developed, the dependencies and relationships between scenarios have to be documented and tested separately. This is achieved by using dependency charts to capture these relationships and dependencies, and by traversing paths in the dependency chart to determine test cases to test the interrelations between scenarios.

REFERENCES

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

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

[3] N. E. Fuchs, U. Schwertel, R. Schwitter, "Attempto Controlled English - Not Just Another Logic Specification Language," Logic-Based Program Synthesis and Transformation, *Eighth International Workshop LOPSTR'98*, Manchester, UK, 1998.

[4] M. Glinz, "An Integrated Formal Method of Scenarios Based on Statecharts," *European Software Engineering Conference*, 1995.

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

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

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

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

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

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

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

[12] 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.

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

[14] J. Ryser, M. Glinz, "SCENT: A Method Employing Scenarios to Systematically Derive Test Cases for System Test," Universität Zürich, Institut für Informatik, Zürich, *Berichte des Instituts für Informatik 2000/03*, 2000.

[15] 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.