UML-Based Statistical Test Case Generation

Matthias Riebisch, Ilka Philippow, Marco Götze

Ilmenau Technical University, Max-Planck-Ring 14, D-98684 Ilmenau, Germany
{matthias.riebisch|ilka.philippow}@tu-ilmenau.de, marco.goetze@stud.tu-ilmenau.de

Abstract. For incremental iterative software development processes, automated testing is necessary to enable evolution not only in terms of functionality, but in terms of software quality as well. Automation requires models to provide the necessary information. Scenarios and use cases do not only feed requirements engineering, they may also be the basis for testing. They have to be enriched by detailed behavioral information in order to be used for statistical test case generation. This paper introduces an approach for generating system-level test cases based on use case models and refined by state diagrams. These models are transformed into usage models to describe both system behavior and usage. The method is intended for integration into an iterative software development process model. The resulting test cases are suited to be carried out in conventional ways, i.e., either manually or using test tools. The method is supported by an XML-based tool for model transformation.

1 Introduction

Object-oriented modeling techniques using the UML play an important role in commercial software development [18]. Comprehensive testing is considered an obvious prerequisite for high software quality. The definition of test cases is a particularly demanding part of every development process. Currently, there are no established methods for the systematic generation of test cases based on UML models. Increasing the efficiency of obtaining test cases and coverage achieved by test cases, will increase quality and productivity and reduce development time as well as costs. Furthermore, the use of UML models for purposes of generation encourages more extensive modeling. The information described in these models is of great use to requirements specification as well. The result is a threefold benefit: generating test cases with high efficiency and quality, more detailed information for requirements engineering, and – motivated by the more intense exploitation of the models – an encouragement for developers towards more in-depth modeling and precise maintenance of models.

There are different categories of tests:

- methods for black box and white box testing, depending on whether or not the software's internals are known and used as input for defining test cases,
- the distinction of unit, component, and system testing, specifying the level at which testing occurs,
- tests classified by their aim, as distinguished by statistical, reliability-oriented testing vs. testing aimed primarily at fault detection.

For a detailed discussion of testing techniques, see, e.g., [10] and [11].

In this paper, an approach aiming for automated generation of test cases based on UML models is explained. It allows for the systematic transformation of a use case model of a software system into a usage model [19]. The usage model serves as input for automated statistical testing [16]. This way, the approach spans the chasm between requirements engineering and reliability engineering as motivated by [15]. It is characterized by the following features:

- It is usage-oriented and specification-based, thus performing *black box* testing.
- It is intended for *system-level* testing, since a given specification usually describes the system's overall functionality rather than that of units or components in isolation.
- It aims at *statistical* (reliability) testing rather than fault detection.

The approach is based on three major concepts, namely, an iterative, UML-based software process model, usage models, and statistical usage testing. A brief consideration of the basic concepts is featured in Section 2. The method for generating test cases based on UML models is described in Section 3. The approach focuses on state-based, interactive rather than data-processing systems. The method is supported by the tool UsageTester [7] that exchanges model data with other development tools via XML. Currently, the approach is being applied in an industrial project within the insurance domain. During this work, the method is refined and, possibly, extended in order to be applicable to large and complex systems. Remarks concerning this and pointers to works dealing with possible extensions and alternative approaches are provided in Section 4.

2 Fundamental Considerations

In this section, the three major concepts involved in the approach are described briefly. Each concept corresponds to a specific modeling step, as shown in Fig. 1 (This diagram is an extension of the one shown in [16]).

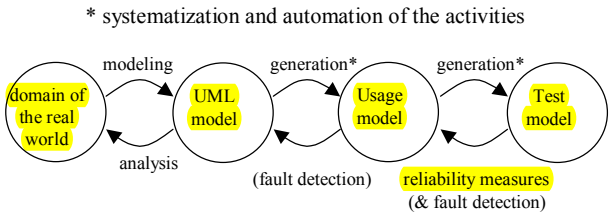


Fig 1. Modeling domains within a software design

The approach described in this document focuses on systematizing or even automating the indicated steps in the process to the maximal extent feasible. (Fault detection is set in parentheses since statistical usage testing focuses on reliability measures and can identify errors only coarsely.)

2.1 Iterative Software Development

During software development, not only the customer's requirements change, but the run-time environment, organizational environment, and legal regulations change, too. Moreover, requirements are often incomplete or partially uncertain, or require elaboration during development. Therefore, the development process needs to be iterative. Both the software architecture and the source code have to be changed frequently to accommodate the changing requirements and constraints.

Integrating the changes without introducing errors is a strong challenge for software developers and project managers alike. Both of them have to ensure that every iteration of the development process leads to progress in terms of functionality without loss in terms of software quality. To control projects, models for software development processes have been introduced. Waterfall-like models have proven inadequate in practice [9]. Therefore, iterative process models, such as the spiral model, have been developed.

Below, a development process model is shown, which:

- has been adapted with respect to the characteristics of object-oriented design using the UML and
- focuses on use cases models and architectural decisions.

The following important properties of software development processes (e.g. [9]) are taken into consideration by this model:

- The processes are iterative and consist of increments or milestones (the progress of projects has to be visible).
- They are evolutionary and should be event-oriented (reacting to new requirements).

An iterative model for software development processes (Fig. 2.) can be represented by a kind of spiral staircase [22]. Each cycle corresponds to a single iteration. A step can be interpreted as an increment. The people involved in the project have different views on a software project, depending on their specific role within the project. In this paper, view-related problems (see [22]) are not discussed in more detail.

The first four steps are use-case-oriented while the last four are architecture-oriented. At each step, a main activity has to be performed. The results of the activities can be described almost completely by UML diagrams. The final targets of main activities are a *detailed use case specification and modelling* and the *definition and modelling of architectural aspects*.

The steps belong to different processes (see Fig. 2). Individual activities of these processes occur either in sequence or in parallel during the software development. During early iterations, activities related to requirements and modelling dominate, whereas in late iterations, activities concerning conversion and introduction are dominant. During all iterations, the modelling process and its results are relevant for:

- comparing requirements with models and model reuse,
- integrating late requirements into models and generating model-based source code, and
- qualifying models during conversion and introduction.

All activities of the development process have to be accompanied by quality assurance activities, such as inspections, reviews, and tests. To ensure the overall efficiency of the process, these activities have to be both highly effective and of low effort.

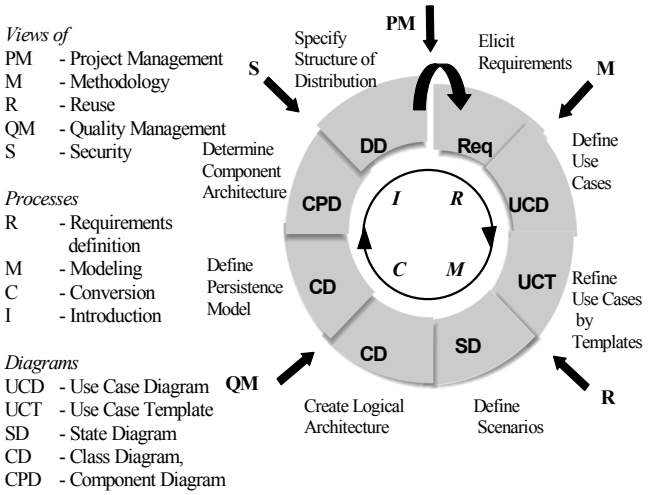


Fig 2. UML-based process model for software development

The UML offers a standardized set of diagrams which can also serve as a basis for systematic (and even partially automated) generation of test cases. This specifically refers to, e.g., use case and state diagrams.

Use case diagrams describe the required functionality of a system in interaction with its environment (external actors) and define its border. A single such diagram comprises several to many scenarios, i.e., possible sequences of interactions between the system and actors. Use cases are often detailed beyond the (limited) scope of use case diagrams using proprietary textual notations, to specify, e.g., exceptions to the "normal" behavior and specify the control flow. Their relevance for testing can be considered as follows:

- Use case diagrams define the required functionality from an external point of view and thus provide information relevant for black box testing.
- Each actor may form the basis for a different usage model and thus different testing criteria (Section 2.2).
- Use case diagrams help determine the frequency of use of specific use cases, knowledge that is required for statistical usage testing (Section 2.2.).

However, use case diagrams are but a rather simple means of formalization and limited regarding the level of detail expressed.

State diagrams are state charts in an adapted Harel notation [8] which describe the behavior of a class or an object. In our approach, we decided to use state diagrams instead of activity diagrams, even though the later are recommended by many UML-based process models for specifying use cases for the following reasons:

- State diagrams comprise all possible scenarios for a given object and appear suitable for direct transformation into a usage model (see Section 2.2). However, unless a single state diagram is used to describe the behavior of an entire component/system, the scope of resulting usage models will be limited to the corresponding class and thus unit testing.
- Forming the basis for comprehensive code generation, state diagrams may be isomorphic to the corresponding program code, facilitating the location of errors found during testing in the dynamic model.

- Again in combination with code generation, state diagrams allow the animation of the state transitions taken during the execution of an executable model and thus (interactive) white box testing.
- As practiced by a number of available tools, state diagrams are the basis for automated model checking (a type of constraint-based white box verification approach), provided that appropriate constraints have been defined.

Summing up, the UML use case and state diagrams are of particular interest in the context of this paper. Other UML diagrams for behavioral specification, such as sequence diagrams and collaboration diagrams, focus on classes and objects, which is why they are incompatible with system-level specification as applied by our approach.

On a side note: albeit rare, efforts have been made to conduct tests of OO-modeled software based on information other than that that depictable by use case and state diagrams. Although not in direct reference to UML, e.g., [5] describes an approach analyzing a software system's OO structure using relational database representations, determining state/transition classes, constructing a data flow graph and applying data flow analysis techniques to that. The approach focuses on integration testing, too.

Use case diagrams will form the starting point of the proposed approach since that way the advantage of *concurrent* software development and test planning provided by a usage model is maintained (see Section 2.2).

We consider the tool-supported generation of test cases based on UML model information as a promising approach. In iterative processes with frequently repeated testing activities set apart by only minor changes, (semi-)automatic generation of test cases and (semi-)automatic testing is of special value.

The activities of the approach described in this paper have been incorporated into the iterative process, as shown in Section 3 (Fig 3).

2.2 Statistical Usage Testing and Usage Models

Statistical usage testing is a system-level technique for ascertaining that software products meet a certain level of reliability ([16], [19]). These and other assessments allow for secondary measures to be determined, such as software readiness and, eventually, a low-risk decision on delivery of the software to the customer.

Statistical usage testing is based on the idea that different parts of a program don't need to be tested with the same thoroughness. There is the (in)famous 90-10 rule stating that typical software spends 90% of the time executing 10% of the code. That means, different portions of software are executed with a higher frequency than others. Statistical usage testing aims to identify these portions and adjust test suites accordingly, subjecting more frequently executed sections of code to more thorough testing.

Since the number of test cases a system can be subjected to is infinite, sampling techniques are required to select an adequate set of test cases. A common sampling technique in the context of statistical usage testing is usage analysis and modeling. Usage modeling defines usage profiles as so-called usage models.

Commonly, only successful scenarios are subjected to statistical testing. However, even negative test cases can be derived by scenarios describing the behavior with invalid input data. These cases can be accommodated by usage modeling as well.

Usage Models: "A software 'usage model' characterizes operational use of a software system. 'Operational use' is the intended use of the software in the intended environment, i.e., the population from which a statistically correct sample of test cases will be drawn." [19]. Usage models use the so-called usage profiles to describe how a software system can be used by different users and how likely different uses in different ways. They are based on the functional and usage specification for software. This information can be acquired even before the implementation has started. As a result, software development and usage modeling efforts become independent, thus allowing for test planning to occur in parallel with or even prior to software development, reducing overall development time and providing additional information for the developer. Fig 6 shows an example.

Usage models resemble finite state machines with transitions weighted with probabilities. The resulting model typically is a Markov chain (although other notations, such as graphs and formal grammars, can be used alternatively) describing the state of usage rather than the state of the system [21].

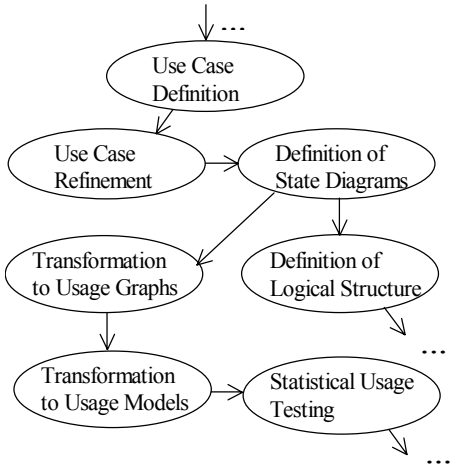


Fig 3. Activities of the proposed approach within the software development process

Markov chains have a unique start and a unique final state representing invocation and termination of the software. Intermediate states resemble usage states interconnected by transitions (unidirectional arcs in a graphical representation). The Markov property requires independence of the next state from all past states given the present state.

Once the structure of a usage model has been determined, probabilities are assigned to all transitions, based on the expected usage of the software. The probabilities of all transitions "leading away" from a state need to add up to 1.

Often there are several categories of users (in UML use case diagrams, these are depicted as actors) a system may interact with, and even several sub-categories per individual type of user identified by secondary criteria, such as experience. A single usage profile typically will not suffice to account for the resulting differences. Instead, several distinct usage models will be created.

A detailed, general methodology for defining usage models of software systems is described in [19].

3 Derivation of Usage and Test Models Based on UML Diagrams

In this section, step-by-step instructions are given describing a possible approach, which systematically leads from given use case diagrams to a basis for automatic generation of test cases. The activity diagram (Fig. 3) shows the sequence of activities in the context of the software development process. For illustration purposes, a simple example taken from a library project is used. A more detailed example discussed in [6].

3.1 Refining Use Cases

Starting out with the definition of use cases, a simplified overall view of the system's required functionality is constructed. Due to expressional limitations, use case diagrams are by themselves of little use – unless the individual use cases are refined in a commonly textual fashion.

The goal of systematically deriving usage models for a software systems requires that dynamic aspects are modeled from a usage-oriented point of view, in particular the system's interaction with the environment, i.e., its response to stimuli; internal reactions are relevant only as long as they yield outbound results.

A well-suited tabular template for textually refining use cases was first published in [2] and has been adopted in, e.g., [3]. The idea behind this textual notation is to obtain a single, complete description of a use case by specifying and mutually relating all of the scenarios it includes. Furthermore, pre- and post-conditions define the usage state before and after, respectively, the execution of the "procedure" the use case resembles. The inclusion of sub-use-cases within a use case allows hierarchical relations among use cases. A version of the use case template (Table 3.1.) extended by actors and invariants is shown in Fig 4; a more illustrative example of a filled-in use case template is given in Fig 5.

The template as introduced in the previously mentioned works needs to be extended for multiple conditions:

- Multiple pre-conditions: pre-conditions define the context, i.e., usage state, in which a use case may be executed; this extension allows for multiple contexts.
- Multiple post-conditions: a single use case can lead to several possible post-states (via variations and/or extensions); the post-conditions may differ depending on which scenario applies. For later use, multiple post-conditions should be numbered, and the scenarios' definitions should make clear to which post-condition they point to.

Table 3.1. Template for refining use cases

Name	identifying the use case.
Goal	describing the overall purpose of the use case
Actors	involved in the use case.
Pre-conditions	needed to be matched in order for the use case to be "executed."
Post-conditions	describing the usage state after the "execution" of the use case.
Invariants	Conditions or state attributes that hold both when the use case starts and throughout its course.
Main Success scenario	Describes how the use case's goal can be achieved as an enumeration of alternating stimuli and responses of the system, starting with the stimulus triggering the use case.
Variations	An alternate course of action which, unlike what is named "extensions" below, is still within what resembles normal parameters for the use case. Variations are specified by referring to the respective step's number in the enumeration of the main success scenario and refining the step with one or more alternative steps. Unless explicitly stated otherwise, the variation replaces the step in question, and the scenario continues with the following step in the main success scenario. Nested variations can be specified by further sub-references.
Extensions	A scenario in response to exceptional circumstances (e.g., invalid input data). Its specification adheres to the same formalism as that of variations described above.
Included use cases	A list of other use cases used by this one (usually those referred to via <<includes>> arcs in use case diagrams).

One such use case template has to be filled in for each of the use cases. The granularity of the scenario specifications defines the granularity of the tests the approach eventually leads to, i.e., a scenario step resembling a stimulus will be turned into an atomic test input and a scenario step describing a response will become an atomic observable response.

3.2 From Use Cases to State Diagrams

During the next step the use case templates are transformed into state diagrams.

The goal of deriving test cases based on usage models requires a more graph-like representation of the usage-oriented view. State diagrams form a suitable intermediate step for two reasons:

- their view is state-oriented and thus suitable to represent different states of usage,
- the prototype of state transitions is a trigger, that is, a stimulus causing the modeled system to change its state.

The introduced use case template has been enhanced to support the systematic derivation of a state-diagram-based usage specification. Detailing the use case is to be done by hand, but the following transformation can be automated by this approach.

Table 3.2. An example of a filled-in use case template

Name	Lend a book
Goal	Lend a book of this library to a user of this library for a limited time.
Actors	Librarian
Pre-conditions	The library system is running. The user is a valid user of the library.
Post-conditions	The book has been lent to the user. A due date for its return has been defined.
Invariants	None.
Main Success scenario	1) The librarian opens the user selection dialog, the selected user is shown. 2) The librarian opens the lending dialog for selecting a book, the return date is shown. 3) He confirms the operation. 4) The system marks the book as lent and adds it to the *lent-book-list* of the user.
Variations	None.
Extensions	2a) The user has to pay a fee or a fine. 2a1) The system displays a message that no new lending procedures are allowed before all due payments have been fulfilled, and switches to the cashing dialog. 2a2) After successful payment, the system returns to the book lending dialog. 2a3) If not successful, the system displays the user selection dialog. 2b) The book of interest is already lent or is for reference only. 2b1) The system displays a message and returns to the book selection dialog. 3a) The librarian cancels the operation. 3a1) The system returns to the book selection dialog.
Included use cases	Display books lent by a user; Collect payment from a user.

The idea (which bears some resemblance to the approach discussed in [3]) is to model each use case in a separate state diagram and anchor those diagrams in a top-level diagram. This top-level diagram resembles a framework in which use cases can be "executed" depending on their pre-conditions, and in which additional global usage states, extrapolated from all of the use cases' pre-conditions, can be switched among, again by means of use cases. A top-level state diagram (or a hierarchy of such, for more complicated systems) is constructed by adhering to the following guidelines:

- From all pre- and post-conditions of all use cases, global states of usage are extrapolated. In this context, "states of usage" refers to the user-system relation. For example, in an interactive GUI-based application, such states might represent different menus.
- Use cases' state diagrams appear as placeholders, meaning that only their state border and name are included, referencing the actual state diagram. Inside these, diagram connectors corresponding to the post-conditions are drawn. A diagram connector is a symbolic reference to another occurrence of the same symbol in another diagram, allowing for connections among separate diagrams. This mechanism is an extension of UML state diagrams. Connectors allow separate state diagrams per use case rather than one complex diagram and benefit modularization/reuse (improving readability and avoiding mistakes). Variations and extensions of use cases leading to additional post-conditions in refined state diagrams frequently supply additional diagram connectors.

- Each of the use cases' place-holding states receives inbound transitions from all of the global states representing pre-conditions of the use case in question. The trigger of each transition is the main success scenario's first step (i.e., the stimulus triggering the use case's "execution"). If there are multiple triggering stimuli, a separate transition needs to be inserted for each of the stimuli.
- The diagram connectors within the use cases' place-holding states are to be connected to the global system states corresponding to the post-condition resembled by the diagram connector in question. These transitions are automatic transitions (and will be subject to further transformation later on).
- Initial and terminating transitions are added, representing the software's activation and termination respectively.

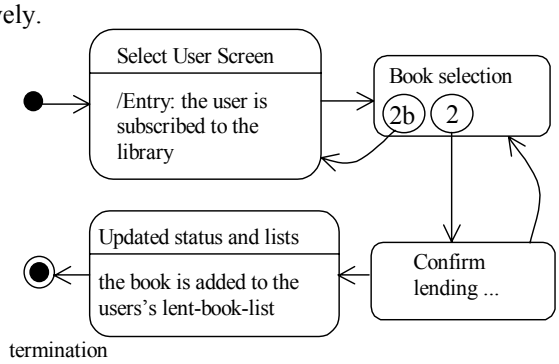


Fig 4. Top-level state diagram for the use case example of Table 3.2. The diagram connector with number 2b has been used to represent the extension 2b

To account for different actors and thus different usage profiles, it is recommended to create a separate top-level diagram for each actor. This way, the independent usage profiles that are represented by the separate top-level state diagrams can be refined independently, retaining and building upon the common specification derived during the previous steps (see Section 3.4). Fig 4 shows a possible top-level diagram for an incomplete pseudo-application the sample use case in table 3.2 might belong to.

Drawing on its filled-in use case template, each of the use cases is transformed into an individual state diagram by following the guidelines below:

- Modeling the main success scenario as state diagram: System responses are turned into (flat) states, stimuli become triggered transitions. States and transitions are named and labeled in a fashion that helps correlate them with the corresponding entries in the respective use case's template,
- Variations and extensions are incorporated by adding more states and transitions, possibly in subordinate state diagrams.
- Each of the different post-conditions becomes a diagram connector with which only inbound transitions are allowed to be connected. Note that it is legal to have multiple transitions connect to the same diagram connector since these markers will at a later stage be replaced by higher-level states.
- Pre-conditions are not modeled inside the respective use case's state diagram, and neither are stimuli that are main scenarios' first steps (unless there are variations and/or exceptions

of the first step in the main success scenario, in which case the diagram's initial transition may be split using a conditional fork).

- Observable system actions are modeled as (entry) actions in the corresponding states.

Fig 5 shows the refined state diagram for the library example. The diagram connectors refer to the diagram of Fig 4 and to the extensions in Table 3.2.

The result of this step is a hierarchy of state diagrams and a higher degree of formalization. The use cases are now tied together via a top-level state diagram. Another advantage of using state diagrams as an intermediate step consists in that, if desired and feasible, given the structure of the system, separate usage models could be created per state diagram, allowing for application of the approach at a component rather than the system level.

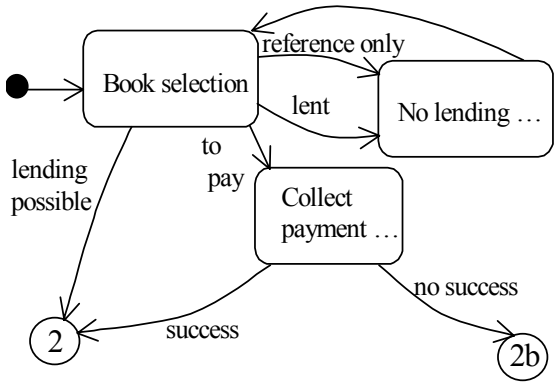


Fig 5. Refined state diagram for the use case example of Table 3.2

3.3 From State Diagrams to Usage Graphs

In this step, top-level state diagrams are transformed into usage graphs. A usage graph has a single starting and a single final state. Further states represent usage states. Transitions between states are labeled with the user action causing the transition to be taken or, as an extension of the general concept, ε. Epsilon transitions are transitions taken automatically, introduced to allow transitions between usage states without external stimulus (for a more detailed discussion, see [6]). The necessity of their introduction will become apparent below.

Top-level state diagrams can be transformed into usage graphs adhering to the following guidelines:

- Flattening of state diagrams is done by recursively including subordinate state diagrams which have so far been substituted by placeholders. The diagram connectors introduced as placeholders for higher-level states corresponding to a certain post-condition are replaced by the higher-level states they symbolize. Subordinate states' initial transitions are replaced by the arc(s) leading to the respective state in the higher-level diagram. State labels should be iteratively prefixed with the state diagram's name (as determined from the corresponding use case) to avoid name clashes. The result will be a state diagram without hierarchical states and diagram connectors.

- Stimuli are unified, i.e., all occurrences of the same stimulus are replaced by a symbol uniquely representing that stimulus. This leads to a more readable graph.
- Automatic (i.e., unconditional) transitions are replaced by transitions with ε as trigger. The alternative would be to remove those states from which automatic transitions extend and assign their inbound transitions their state actions as transitional actions. The epsilon approach enhances readability and correlation of the usage graph with the state diagrams it was based on. Besides, it avoids merging the modules represented by state diagrams and thus blurring their borders.
- The initial transition of the top-level diagram is replaced by a state named *Software Not Invoked*, resembling the as yet uninvoked (part of the) application, and a transition leading from there to the target of the original initial transition. The new initial transition is assigned a dedicated symbol representing activation of the (part of the) software.
- The terminal state is replaced by a state named *Software Terminated* in a way equivalent to that specified for the initial transition. Likewise, a symbol resembling termination of the software is assigned to the respective new transitions.
- There must be no dead-end (i.e., states without any outgoing transitions) or unreachable states. If any are found, iterative revisions will be required.
- State actions are kept in place. Since the labeling of usage graphs' states is arbitrary, they may be considered a part of a name or comment and in any case allow for later identification of the expected system output when a certain usage state is reached.

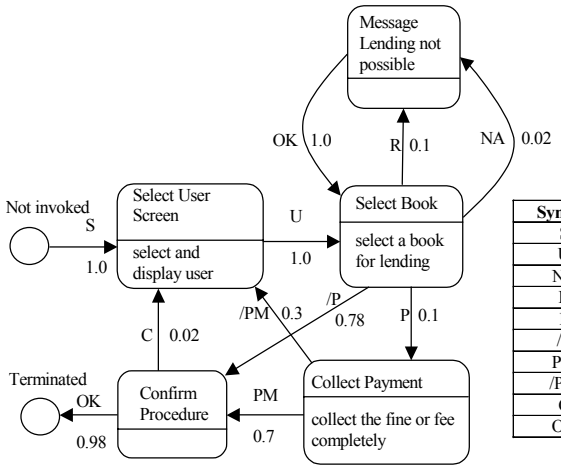
An illustration of how a usage graph of the example used above might look like is given (as a completed usage model derived without further modifications from the usage graph) in the following Section.

3.4 From Usage Graphs to Usage Models

To obtain usage models the probability distribution of the expected use of the software with respect to the usage profile represented by the usage graph must be determined. There is no general systematic approach for deriving that. It needs to be determined in conscious evaluation of the modeled usage graph, the given specification, discussion with the customer, etc.

The probability of some outbound transition of a usage state may be found to vary depending on previous uses of the system. Since this is in violation of a usage model's Markov property, the state in question and all dependent states need to be replicated and connected so that historic dependencies are resolved and, eventually, all of the usage graph's states' invocation probabilities depend only on the neighboring preceding state. Additionally, different subcategories of users represented by a single actor may need to be distinguished by criteria such as experience.

This would make it recommendable to fork off separate usage models at this point, each refined and weighted according to the probability distribution most appropriate for the user profile in question. For guidelines concerning the commonly iterative cycle of assigning transition probabilities, see [19]. An approach for systematically determining transitions probabilities using objective functions derived from structural and usage constraints is introduced in [20]. As a basic means of verification, the sum of all transitions originating from one state must be equal to 1. The (outbound) transition of the state *Software Not Invoked* is taken with probability 1, and so are epsilon transitions. Each state must have a next state (the next state of *Software Terminated* implicitly is *Software Not Invoked*).



Symbol	Description
S	Select lending books
U	User selected
NA	Book not available
R	Book for reference only
P	User has to pay a fee or fine
/P	No payment needed, lending possible
PM	Payment successful
/PM	Payment not successful
C	Cancelled by librarian
OK	Confirmed by librarian

Fig 6. Possible usage model for the library example of Table 3.2. The usage model has been derived without modifications from a usage graph created in adherence to the guidelines given in section 3.3., 3.4

3.5 From Usage Models to Test Cases

Based on a usage model, it is possible

- to produce data that can be used throughout the project's life cycle [21] for test planning and
- to derive statistically valid samples of test cases.

Usage models are predestined for random testing. The basic prerequisite for random testing and initial test suite is the minimal arc coverage suite (see [14]): a sequence of transitions traversing the model in as small as possible a number of steps, a sequence which can be determined automatically. During the determination of the sequence as well as during the generation of random walks later on, epsilon (ε) transitions can be silently ignored (i.e., left out from the resulting sequence of stimuli) since they are taken automatically by the software, and their coverage is ensured by covering all other arcs (since all states are reachable (ensured in turn by assigning only probabilities greater than zero)).

For instance, a minimal arc coverage suite for the example used in this chapter (Fig 6) is S U R OK NA OK P /PM U P PM C U /P OK achieved by a single run of the application.

After the minimal arc coverage suite has been passed, random test cases are generated. Each test case is a random traversion of the usage model with the choice of which exit transition is taken at any given state determined by the probabilities attached to the states' arcs. A random test case for the example might, e.g., be resembled by the sequence S U P /PM U P PM OK.

During the execution of test cases, the correctness of the behavior of the software in response to events in the sequence is recorded as either "pass" or "failure," in the later case also recording the test case number and transition that led to the failure. Depending on the severity of failure, it can be treated as a "stop" or "continue" failure, causing the test case to be either

stopped or continued, respectively. Failure data and usage model analyses can be used in conjunction to produce metrics for test sufficiency and product quality [21].

Two basic measures for test sufficiency are resembled by state and arc coverage. Both coverage criteria are determined by the probability distribution in the model. Additionally, a determinant can be computed reflecting the similarity between expected and tested use [22]. It is up to the test engineer to decide what threshold value it needs to have reached before testing is considered sufficient.

Other measures obtainable from test results concern product quality, expressed as, e.g., the Mean Time Between Failures (MTBF).

4 Related Works

While incompatible with this approach and outside the scope of this document, a number of works dealing with the generation of test cases based on UML state diagrams have been published and shall briefly be described and referenced below. Note that none of these works use state diagrams to derive usage models, but rather all of them focus on obtaining test cases directly from state diagrams.

- [15] motivates the combination of scenario-based requirements engineering with verification and validation. It discusses benefits and strategies. Our approach adheres to the recommended transformation strategy.
- [3] inspired this work by employing use case templates as described before to refine use cases and transforming these into state diagrams. The further procedure, however, involves formulating and solving a STRIPS [4] planning problem based on the state diagrams.
- The approach described in [10] flattens and transforms state diagrams into extended finite state machines (EFSMs) and in turn transforms those into flow graphs to which conventional data flow analysis techniques can be applied.
- [1] describes an approach transforming state diagrams into transition tables, enumerating transition predicates, and deducing test cases satisfying various coverage levels.

An alternative usage-centered approach to testing without reference to UML notations is introduced in [14]. It starts out with given textual requirements for a software system, derives canonical sequences of stimuli, and constructs a usage model from those.

5 Conclusions

The described procedure resembles a baseline approach to the goal of automated generation of test cases based on UML models. By combining scenario-based requirement descriptions with system behavior descriptions for test case generation, the approach offers three major benefits. Firstly, it increases the efficiency with which test cases are generated. Based on this fact, it supports iterative development processes by reducing testing efforts. Secondly, it presents a way of refining use case diagrams using state diagrams and usage information thus, supporting requirements engineering. Thirdly, it encourages developers to develop more detailed models by offering (potential) tool support in both modeling and model-based generation.

An additional benefit of applying the approach aside from obtaining test cases systematically, is its rather formal and (in the authors' opinion) easily comprehensible documentation of the software's usage and interactive behavior. This fact can be considered a slight reduction of the overhead testing imposes on the software design process. Furthermore, the resulting refinement shall be the first step of defining a fine-grained model of system behavior, leading to a methodology of model-based software construction as an aim of newer works.

Even though the approach described is UML-based primarily in its evaluation of use case diagrams, it uses state diagrams as a UML notation to facilitate the creation of usage models. The usage-oriented nature of usage models and statistical usage testing, limits the extent to which information relevant to testing can be derived from state diagrams created during the actual development process of the software. In the authors' opinion, it is still justified to consider the approach UML-based since use case diagrams and refinement of use cases obviously are integral aspects of a UML-based development process.

The approach described in this paper is implemented in the tool UsageTester [7] to provide a proof of concept and to support the method's application. As a result of current industrial projects, further refinements will certainly be necessary. In the following, some potential issues are listed:

- Transforming refined use cases into state diagrams and extrapolating a global framework of additional states resembling user-system configurations should work well for single-threaded applications. However, dealing with concurrencies will complicate the process and require refinement and, possibly, extensions of the procedure.
- The Markov property of usage models requires that the next state in a sequence of interactions depend only on the present state. This means that different software states for the same usage state must not cause the next usage state to be different. If such dependencies exist in a software subjected to usage modeling as described in the approach, the internal state and its consequences need to be propagated to the usage view of the system, i.e., considered from the process of refining use cases onward. How such dependencies can be detected and appropriately incorporated will require further analysis and refinement of the described approach.
- The procedures for extrapolating additional usage states to add to top-level state diagrams and separating initial and final steps in a scenario (as detailed using the use case template) from pre- and post-conditions, respectively, need to be refined and formalized. Although the approach is not completely automatable, it should at least be automatable in part. Obviously, the generation of random test cases can easily be automated. Furthermore, the transformational steps leading from state diagrams to usage graphs and the replication of sub-graphs of a usage model in order to eliminate run-time history dependencies should be possible to be aided by software. The details of how these automations might be implemented will require further analysis.
- Flattening state diagrams leads to complex structures. For cases exceeding current limits, methods for hierarchically structuring usage profiles have to be developed.

6 Acknowledgements

We wish to thank Prof Gwendolyn H. Walton from UCF Orlando for fruitful discussions about the role of statistical testing.

References

1. Abdurazik, A., Offutt, J.: Generating test cases from UML specifications, In: Proc. 2nd International Conference on the Unified Modeling Language (UML99) , Fort Collins, CO, October (1999)
2. Cockburn, A.: Structuring use cases with goals. Journal of Object-Oriented Programming, Sep/Oct, Nov/Dec (1999): 35–40, 56–62, resp., (1997).
3. Fröhlich, P., Link, J.: Automated test case generation from dynamic models. In: Proc. ECOOP 2000, LNCS 1850, Springer (2000) 472–491.
4. Fikes, R. E., Nilsson, N. J.: Strips: A new approach to the application of theorem proving to problem solving. Artificial Intelligence 2 (1971).
5. Gallagher, L.: Conformance testing of object-oriented components specified by state/transition classes, Draft technical report. NIST (1999) Available online at http://www.itl.nist.gov/div897/ctg/stat/auto/autosys.pdf
6. Goetze, M.: Statistical Usage Testing based on UML Diagrams. Student's Work. Ilmenau Technical University, Dept. Process Informatics, (2001)
7. Goetze, M: UsageTester: UML-oriented Usage Testing. Ilmenau Technical University, Dept. Process Informatics, (2002)
8. Harel, D.: Statecharts: a visual formalism for complex systems. Science of Computer Programming, 8 (1987) 231–274
9. Jacobson, I., Griss, M., Jonsson, P.: Software Reuse - Architecture, Process and Organization for Business Success, Addison Wesley (1997).
10. Kim, Y. G., Hong, H. S., Cho, S. M., Bae, D. H., Cha, S. D.; Test cases generation from UML state diagrams. IEEE Software, 146(4), (1999) 187–192
11. Marick, B.: The Craft of Software Testing. Prentice Hall (1995)
12. Myers, J.G.: The Art of Software Testing. John Wiley & Sons, Inc. (1979)
13. OMG: UML Notation Guide, v1.3. Object Management Group, Inc. (2000)
14. Prowell, S.J., Trammell, C.J., Linger, R.C., Poore, J.H.: Cleanroom Software Engineering. Addison-Wesley, 1st edition, (1999)
15. Regnell, B., Runeson, P.: Combining Scenario-based Requirements with Static Verification and Dynamic Testing. Proc. Fourth Intern. Workshop on Requirements Engineering: Foundation for Software Quality (REFSQ98), Pisa, Italy (1998)
16. Selvidge, J.M.: Statistical usage testing: Expanding the ability of testing. (1999)
17. Thulasiraman, K., Swamy, M. N. S.: Graphs: Theory and Algorithms. John Wiley & Sons, Inc. (1992)
18. UML Ressource Page. Available online at http://www.omg.org/
19. Walton, G.H., Poore, J.H.: Statistical testing of software based on a usage model. Software - Practice and Experience, 25 (1995) 97–108
20. Walton, G.H., Poore, J.H.: Generating transition probabilities to support model-based software testing. Software - Practice and Experience, 30 (2000) 1095–1106
21. Whittaker, J. A., Thomason, M. G.: A Markov chain model for statistical software testing. IEEE Transactions on Software Engineering, 20(10): (1994) 812–824
22. Wolf, M., Burkhardt, R., Philippow, I.: Software Engineering Process with UML. In: Schader, M., Korthaus, A. (Eds.): The Unified Modeling Language –Technical Aspects and Applications. Physica Heidelberg, (1998) 271-289