Statistical Usage Testing Based on UML

Michael HÜBNER, Ilka PHILIPPOW, Matthias RIEBISCH
Dept. of Computer Science, Technical University of Ilmenau
98693 Ilmenau, Germany
{Michael.Huebner, Ilka.Philippow, Matthias.Riebisch}@TU-Ilmenau.de

ABSTRACT

This paper shows a way to derive test cases for system level black-box-testing from the specification models already elaborated in the requirements analysis phase.

The basis for this process is the UML (Unified Modelling Language) use case model. It provides a good way to describe both the interaction with the user and the system behavior.

The concept of a text template driven structure editor is presented. Such an editor can be used to construct a formalised use case description in a user-friendly way.

According to the principles of the – already known – statistical usage testing, which aims at a statement about the fitness of the system for the intended purpose, the most likely usage scenarios are chosen as test cases.

It is shown how the Marcov property of the system description can be preserved in the case of data dependent system behaviour.

Keywords: statistical usage testing, usage based testing, behaviour specification, use case, UML, Marcov chain

INTRODUCTION

Extensive and efficient testing is very important to ensure software quality. In many projects the costs for testing represent 25-50% of the overall project costs. It can be said that testing is not a very favoured task. The first step to reduce the effort for testing is to use testing tools that execute tests automatically. But the necessary test cases are usually created manually requiring the tester to think about the usage and the behaviour of the system, a task he or another person has already done in the requirements analysis phase of the software development.

This doubled work can be avoided when the test cases for black-box-testing are derive from the use case models and the class models. These models are available as products of the requirements analysis phase of the software development[1] [1]

During the development of a software system there are other kinds of testing that have to be carried out. Usually, these tests examine just parts of the system and take internal details into account or they are intended to find the location of errors. These tests are outside the scope of this paper because the test cases for these tests can not be derived from specification models alone.

Taking the specification models as the basis for tests has the positive side effect that more attention is paid to keep the models complete and up-to-date. Another advantage is that testing can start in very early phases of the development process which is important for incremental development and allows to shorten the time to delivery. Furthermore the software quality is raised because the system is tested with respect to the explicitly stated user requirements.

Statistical Usage Testing

The basis for creating test cases in statistical usage testing is a *usage model.* A usage model is like a state machine, i.e. it is a directed *usage graph* consisting of states and transitions, with the extension that every state transition is attributed with the probability that this transition will be traversed when the system is in the state from which the transition arc starts. Hence for every state the probabilities of outgoing transitions sum up to one. Every transition can be related to an event (possibly with parameters) which triggers that transition.

A transition with an associated event may also be related to a guard condition. This means that the transition is only performed if the condition is fulfilled by the event parameter value(s).

There are three approaches to assign transition probabilities. In the *uninformed* approach all exit arcs of a state have the same probability. The *informed* approach uses sample user event sequences captured from a prototype or a prior version of the system to calculate suitable probabilities. The *intended* approach allows to model hypothetic users or to shift the test focus to certain states or transitions.

The *Marcov property* states that all transition probabilities depend only on the actual state and are independent of the history. This means that they must be fixed numbers. A system with this property is called a *Marcov chain*, for which some valuable analytical descriptions can be concluded.[2]

One such description is the *usage distribution* stating the steady-state probability for every state, i.e. the expected appearance rate of that state. Since each state is associated with some part of the actual software, the usage distribution shows which parts of the software get the most attention from the test cases.

Other important descriptions are the expected test case length and the number of test cases that are necessary to verify the required reliability of the system. [2] [3]

The idea of generating usage models for statistical testing from use cases is outlined in [4]. We extend this approach by allowing non-deterministic system behaviour and by handling guards that depend on system data state. This enables to apply statistical usage based testing to a wider range of systems.

System Specification with UML

The requirements analysis phase yields at least two models to describe the planned system from the user perspective:

1. the use case model to describe the system behaviour [5]
2. the domain class model to describe which kind of real-life objects are represented in the system and what of their at-

1We choose UML (Unified Modelling Language) for (object-oriented) system modelling because UML is widely used.

tributes, operations and relations (i.e. links) to other objects are relevant for the intended purpose of the system.[1]

This distinction between dynamic and static aspects in the system description is carried over to the notion of state. The overall state of a system is made up of the *execution state* which tells what step of a use case is currently being executed and of the *data state* which tells what data are stored in the system. The data state is made up of the *system data state* which is persistent between use case executions and the *use case data state* which is local to the executed use case and hence transient.

The UML standard defines only a very top-level structure to describe use cases which allows only to define use cases as named entities with a textual description. It's common to write use case descriptions in a slightly more structured form, e.g. in tabular or tree form.[6] Because the leave entries in these structures are still plain text they are *textual use case descriptions.*

EXAMPLE: Parking ticket vendor machine

In this chapter a small example is introduced that will be used to demonstrate the application of our method. It describes a vending machine situated on a parking place. A person who wants to park his car there must buy a parking ticket for the planned parking time from this machine. We give UML specification models for this machine as the starting point of the process.

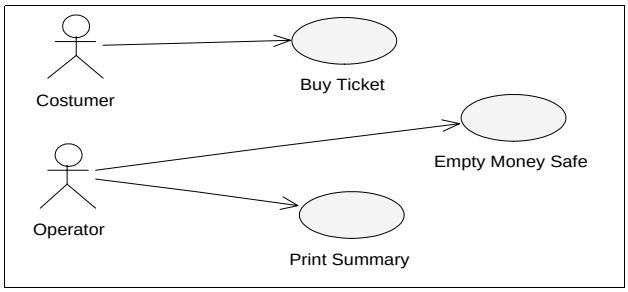

Figure 1: Use case overview

USE CASE	Buy Parking Ticket
Brief Description	*This use case allows a customer to buy a parking ticket. Payment is done by inserting coins.*
Precondition	*System is ready. The coin buffer is empty.*
Postcondition	*The customer has received a parking ticket.* *The inserted coins are moved to the safe.* *System is ready. The coin buffer is empty.*
Exception	*The transaction is cancelled by the customer.*
Exception Postcondition	*The inserted coins are returned to the customer.* *System is ready. The coin buffer is empty.*
Actors	*Customer*
Trigger	*First coin inserted*

Table 1: Tabular use case Description example (part 1)

	Main success scenario
1	*The Customer inserts coins. After every coin insertion, the system displays the updated parking end time according to the amount of inserted money.*
2	*When the Customer is pleased with the parking end time, he requests the ticket by pressing the OK-Button. The System prints the ticket showing start time and end time. The inserted coins are moved to the safe.*

Table 2: Tabular use case description example (part 2)

In Figure 1 there are two user roles with their possible use cases. The tabular description of the use case “Buy Ticket” is provided in Table 1 and Table 2.

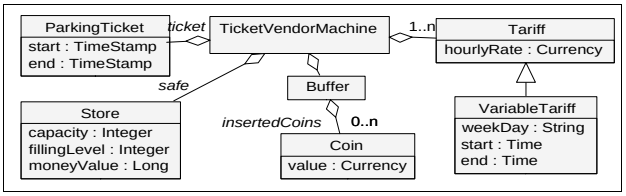

Figure 2: Simplified domain class model example

The domain class model shown in Figure 2 contains elements which are explicitly or implicitly referenced in the textual use case description.

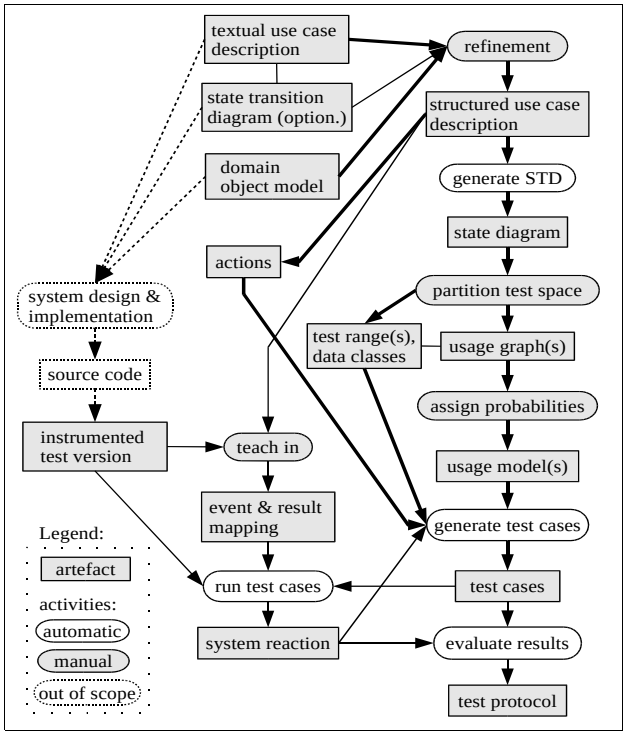

Figure 3: Process overview

APPROACH

Figure 3 shows the artefacts (i.e. models, documents, data) that are used or created in the presented process. It shows the activities leading from the specification models to the test protocol. This protocol is the basis for estimating the reliability.

The presented process consists of the following activities:

1. Refine all use cases – by using the object model - into a structured form
2. Use the refined use case to
   - Derive the usage model
     1. Transform structured use cases automatically into state diagrams
     2. Transform state diagrams into a usage graph for each test range
     3. Enhance the usage graph by a set of probability assignments to get a usage model
   - Map descriptive elements in the structured use case to particular real elements in the system implementation
3. Generate test cases based on the usage model – using the specified behaviour of (data changing) operations performed by the system
4. Run the test and evaluate the results

Textual use case descriptions (as used in Table 1 and Table 2) are well suited for communication with a customer, but they are not formal enough to be amenable for automatic processing to generate test cases. Hence they have to be refined. This first step is further explained in the next chapter.

In these refined use case descriptions, we make use of the UML domain class model, i.e. elements of the class model may be referenced in use case descriptions.

For describing the effect of system actions with respect to the data state we use pre-conditions and post-conditions. These conditions may contain constraints about the existence of objects, about the values of attributes and about the existence of links between objects.

An instrumented version of the implemented system is used to map the descriptive elements (e.g. events, parameters, response) contained in the use case models to the concrete elements of the real system (e.g. button clicks, data entry fields, displayed information). This mapping is the key to automate the test execution.

To create test cases, at any point in a test scenario the expected data state of the tested system has to be known. This is done by keeping the state in an external generic database which is updated according to data changing system behaviour defined in the structured use case description.

Normally, internal data variables of the implementation of the system under test are not mapped to elements of the domain class model, although in some cases this may be helpful to check pre- and post-conditions on system data state.

In the case of non-deterministic system behaviour, the response and the system state can not be pre-calculated. It can only be described by a set of possible states and value constraints. Hence, the sequence of all test cases can not be created in advance. Instead every event has to be determined in real-time. This means that step 3 and 4 of the process are then performed together.

USE CASE REFINEMENT

The most challenging step during this whole process is the first one: the refinement of the use cases. The UML meta-model (which describes the kind of model elements available in UML models and diagrams) has only a few classes to describe use case in a formal way. The behaviour of the use case , i.e. the interaction of a system with its environment, can only be described in an informal way as plain text.

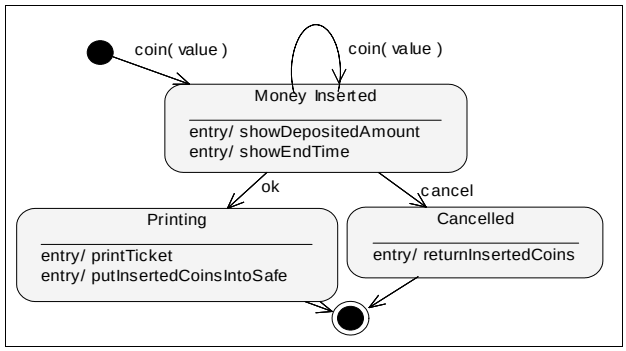

Figure 4: State chart diagram for "Buy Ticket"

UML allows the behaviour of a use case to be described much more formally by accompanying diagrams, like sequence diagrams, state chart diagrams (e.g. like in Figure 4) or activity diagrams.

The informal textual behaviour description cannot be processed automatically to generate test cases. On the other hand, the above mentioned more formal UML diagrams are not understandable by every domain expert. Thus a combination of both forms is needed. The result of refining the behavioural information of a use case is a semi-formal representation consisting of the following:

1. an internal model (usable for automatic processing) – represented as a set of connected description objects according to a meta model – which reflects the intended behaviour of the use case as close as possible and
2. a structured text (for communication with domain experts), which is connected to the internal model by predefined text templates.

The refinement is supported by a specialised editor which allows putting together text blocks from the set of available text templates, causing the internal model to be updated accordingly.

Model elements

The available model elements define the kind of systems that can be described in a manner, that is detailed enough to generate meaningful test cases.

We use model elements to describe:

- the pre- and post-conditions that guide the applicability of the use case,
- the possible sequences of steps
  - guard
  - loop
- the details of every use case step which are relevant for the generation of test cases:
  - user caused events, e.g.
    - data value entry
    - selection
    - button click
  - system response, e.g.

    - display of data values
    - display of a selection list
    - display of a form
- data changing operations
    - create/delete an object
    - set/read a data value
    - add/remove a link between two objects
- (local) use case variables

There are different kinds of pre- and post-conditions. One kind describes execution states of the system, e.g. "*The system is ready*". Another kind describes data constraints, like "*The coin buffer is empty*". This form is expressed into OCL, the Object Constraint Language of the UML. Both kinds of constraints can be combined (as in the precondition of the example use case).

There is another kind of post-conditions which summarise the overall effect of a use case by making statements about the system reaction (possibly depending on the user stimuli), e.g. "*The inserted coins are returned to the customer*". Constraints of this kind can be ignored because their information content is already contained in the use case step descriptions.

	Main success scenario
1	*("**The customer**"{costumer} ([id1:]("**inserts**" "**coins**" [coin>>insertedCoins])){loop}.) (("**After every**" "**coin insertion**"[->id1]) "**the system**"{system} "**displays**" (("**the amount of inserted money**" [depositedMoney=insertedCoins -> sum(value)]) "**and**" ("**the updated**" "**parking end time**"[ticket.end])) .)*
2	*(("**When the customer is pleased with the parking end time,**"{goal}) "**he**"{costumer} ("**requests the ticket**"{goal} "**by pressing**" "**the OK-Button**").) ("**The System**"{system} "**prints**" ("**the ticket**"[ticket] "**showing**" ("**start time**"[start] "**and**" "**end time**"[end])).) ("**The inserted coins**"[insertedCoins=buffer.content] "**are moved to**" "**the safe**"[safe].)*

Table 3: Refined use case description example (part 2) – (in a textual debug representation)

An impression of the structure resulting from the refinement is given in Table 3. It is only one possible representation chosen with the goal to be very concise while showing the underlying structure. The used syntax has the following semantics:

- *"automatically adapted textual representation for a description element"*
- *[ID:](E) – label ID assigned to the following element E*
- *"R"[->id] – R is a reference to the element labelled ID*
- *"T"[X] – name for a variable following its textual representation T*
- *"T"[X>>S] – inside a loop: the values that variable x has in the iterations are accumulated into set S*
- *[X=OCL_expression] – derived data element X with value expressed in OCL*
- *"T"{Kind} – Kind describes a predefined element following its textual representation*
- *"explaining text with no effect on system behaviour" {goal}*
- *(S){loop} – element S can be executed multiple times*

The full XML representation of this structure can be found (together with the used text templates) under [7].

Levels of use cases

There are different abstraction levels for identifying and describing use cases [6]:

- on the very low, technical *interaction level*, every button-click made by a user is described
- on the more abstract *semantic level*, the events initiated by a user are more problem-/ task-oriented. These logical events summarise some related low-level events and hence abstract away unnecessary details.

For real-time systems, the interaction level seems quite appropriate. For complex data-processing applications the semantic level is the better alternative.

The parking ticket vendor machine is described with interaction level use cases.

A system that is better described by semantic level use cases is the following:

Group date book example: A group date book system is used to make appointments for some attendees. For this reason it has the use case "makeAppointment" where events are setAttendees(Set<Person>) and showAvailableTimeSlots(duration, latestDate).

Event parameters and data state

In semantic level use case descriptions of data-processing applications, events often carry parameters (e.g. input data). The system responses may have parameters, too (e.g. the content of the search result list).

The value of event parameters (e.g. a search parameter or a display options) can affect the response, i.e. the observable system reaction. It could only influence the value of response parameters or it could influence which transition is taken. Often a parameter can also be used to change the system data state.

There are also parameters that are both data changing and transition determining. A common example is a parameter that is used as the unique name for a newly created object (e.g. the identification code for a new member of the working group). The transition depends on whether the intended value for the unique name is already used.

Even the interaction level use cases for the example parking ticket vendor machine makes use of parameters in events and responses. Without parameters, separate event arcs for every type of coin had to be used, which would make the model more complex. Another problem is that the allowed coin values are unspecified in the original use case text.

FROM REFINED USE CASES TO STATE MACHINES

The model elements of a refined use case are transformed into model elements in a state machine in the following way:

- Every use case step becomes a state.
- Every user caused event is transformed into an event in the state machine with an associated a state transition.
- Loops are represented by recurring transitions.

FROM STATE MACHINES TO USAGE GRAPHS

State machines and usage graphs are very similar. Hence most model elements of a state machine can be transformed straight forward into corresponding elements in the usage graph.

But transition guards depending on data state (i.e. on system data state or on use case variables) which are valid elements of UML state diagrams have no direct counter-part in usage graphs and usage models. That's because data dependent guards would violate the Marcov property which states that the transition probability may only depend on the actual execution state and not on the history of previous states. It's just this history that is accumulated in the data state.

Handling data dependency by state expansion

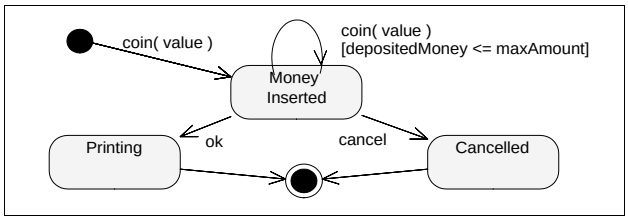

Figure 5: data dependent transition guard

Figure 5 is a variation of the state machine in Figure 4. Here the recursive transition coin(value) has a guard stating that the amount of inserted money may not exceed a given maximal amount. This means that the transition guard depends on the data state, more precisely on the use case variable depositedMoney.

If some aspects of the data state (i.e. some variables or thereof derived values) in the state machine has an influence on the transition probabilities, then the influencing aspects of the data state are simply translated into execution states in the usage model by expanding the original states of the state machine.

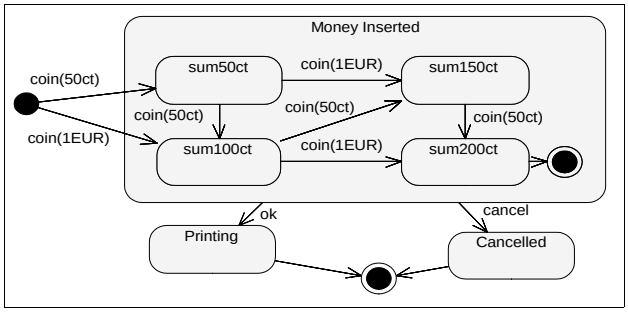

Figure 6: Sub-states to model maximal amount

Figure 6 shows the expanding of state Money Inserted into several sub-states to model the transition guard depending on the use case variable depositedMoney.

In order to avoid a state explosion for the usage graph state expansion is only applicable for a small number of independent aspects, each with a small number of possible values.

Handling data dependency by randomising

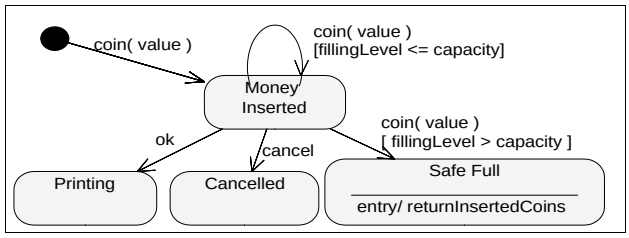

Figure 7

If we consider that the parking ticket vendor machine has a money safe with a limited capacity, then the corresponding variable fillingLevel may have a several thousand possible values. This means that state expansion is not useful.

One solution were to assign the probability 1/safeCapacity to the transition that leads to state Safe Full. Because the deterministic behaviour is described by a non-deterministic behaviour, this approach is called *randomising*.

The randomising of transitions is only needed to derive the analytical description from the resulting Marcov chain. But to generate the real test cases the original guards are used.

The problem with this approach is that the test cases generated from this usage model are very long, i.e. they consist of a long series of events to reach the capacity limit. Hence the test execution would be very resource intensive.

Handling data dependency by test space partitioning

Another approach is to think of state nodes as being parameterized by the transition influencing aspects of the data state. Every such independent data aspect constitutes a dimension in a *data state space*.

Then every parameterized state represents a family of similar states which are connected by a family of similar transitions – with the exception of a few transitions which are different.

A way to evade state explosion is to divide the data state space into different test regions to separate the many common states from the few others. This technique is especially useful in the case of system data state that reflect restricted resources with a large capacity (as the money safe).

For every limited resource the test space is divided into two aspect ranges:

1. not-full: here the case of reaching the capacity limit is ignored in the usage model. Test cases are chosen so that the state stays inside the test range
2. nearly-full: the usage model is aware of the capacity limit

The range nearly-full is modeled by a usage graph where the original state is expanded into a few states below the limit. Test cases are created from this model. But the test driver has to bring the system to a state in the nearly-full test range by a sequence of use case executions.

If there are multiple limited resources, combination of the aspect ranges have to be used to get the test region. If there are to many combinations, probabilities for the ranges are used to determine the most likely combinations.

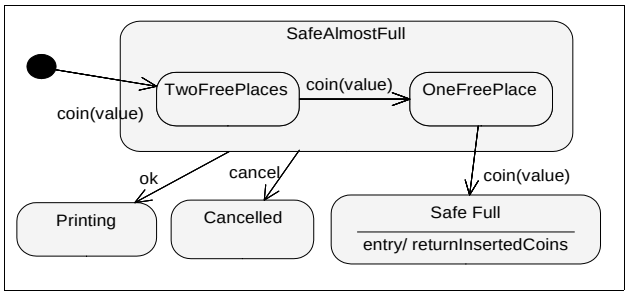

Figure 8: Nearly-full range

The usage graph for range not-full is similar to Figure 4. The usage graph for range nearly-full is shown in Figure 8.

GENERATING TEST CASES

Beginning in the start state the usage model is traversed by selecting transitions according to the specified probabilities. If the transition has no guard, then values for the associated event parameters are also chosen randomly.

If the transitions has a parameter dependent guard, the event parameter values must be chosen so that they fulfill the transition guard condition. For the common case of comparison operations combined with the application of reversible functions this is a straightforward task. For more complicated cases, search strategies could be used.

Interaction between event parameters and data state

Some guard conditions may depend both on event parameters and on system data state. This transition can be assigned a fixed probability if there's a way to compute for every data state a set of event parameters to fulfil the guard.

Example:

In the group date book system one can always find a free time slot if the event parameter latestDate is set to the far future or the planned duration is very short. In this case a simple monotone search strategy can be used to find appropriate parameter values.

CONCLUSION AND FUTURE WORK

We developed and tested the approach based on a variety of examples.

Refined use cases plays two roles. They allow to communicate with non-technical personal via the generated textual representation. On the other hand they are the base for system testing. Refined use cases could even be used to build some kind of prototype of the planned system.

We integrated data dependent transition guards into usage based testing with Marcov chains. This allows to use the analytical descriptions for a wider range of systems.

With the partitioning of the data state space into test regions the efficiency of the test execution could be increased.

For supporting the described process an XML-based tool prototype has been developed which is being enhanced incrementally.

In our ongoing research project we plan to make extension in other relevant directions, like test cases for parallel execution of use cases.

References

[1] OMG, UML resource page , http://www.omg.org/technology/uml/index.htm

[2] J. A. Whittaker and J. H. Poore, Markov Analysis of Software Specifications, *ACM transactions on software egineering and method*, :, 1993

[3] J. A. Whittaker and M. G. Thomason, A Markov chain model for statistical softwae testing, *IEEE Transactions on Software Engineering*, 20(10):812-824, 1994

[4] Björn Regnell, Per Runeson and Claes Wohlin, Towards Integration of Use Case Modelling and Usage-Based Testing, *Journal of Systems and Software*, 50:117-130, 2000

[5] Ivar Jacobson, Object-Oriented Software Engineering - A Use Case Driven Approach, 1992, Addison-Wesley

[6] Alistair Cockburn, Writing Effective Use Cases, 2000, Addison-Wesley

[7] Michael Hübner, Presentation of research project "UML-based test case generation" , http://www.theoinf.tu-ilmenau.de/~huebner/testgen/