Automated Test Case Generation from Dynamic Models

Peter Fröhlich[1] and Johannes Link[2]

[1] ABB Corporate Research Center, Speyerer Straße 4,
D-69115 Heidelberg, Germany
peter.froehlich@de.abb.com
[2] Andrena Objects GmbH, Albert-Nestler-Straße 9, D-76131 Karlsruhe, Germany
johannes.link@andrena.de

Abstract. We have recently shown how use cases can be systematically transformed into UML state charts considering all relevant information from a use case specification, including pre- and postconditions. The resulting state charts can have transitions with conditions and actions, as well as nested states (sub and stub states). The current paper outlines how test suites with a given coverage level can be automatically generated from these state charts. We do so by mapping state chart elements to the *STRIPS* planning language. The application of the state of the art planning tool *graphplan* yields the different test cases as solutions to a planning problem. The test cases (sequences of messages plus test data) can be used for automated or manual software testing on system level.

1 Introduction

The systematic production of high-quality software, which meets its specification, is still a major problem. Although formal specification methods have been around for a long time, only a few safety-critical domains justify the enormous effort of their application. The state of the practice, which relies on testing to force the quality into the product at the end of the development process, is also unsatisfactory. The need for effective test automation adds to this problem, because the creation and maintenance of the testware is a source of inconsistency itself and is becoming a task of comparable complexity as the construction of the code.

To trace requirements throughout the software's life cycle - testing included - seems to be an appropriate way around the above mentioned problems. It can minimise the deviation of the work products from the specification. To maintain consistency throughout the activities of the software process, it has to be defined how each part of the requirements specification impacts each of the downstream artifacts. This strong traceability can already be achieved for certain parts of the requirements. For example, steps in a use case relate to messages in dynamic UML models [19], which map (in some contexts) to methods of a programming language and to steps in a test case. For other parts, the systematic exploitation of the information throughout all phases in the software process is not yet defined.

In the current paper, we describe a full mapping of all elements of a typical use case document [3] to a UML state machine. Based on this state machine we afterwards apply AI planning methods to derive test suites with a given coverage level. The test generation method takes into account the specific elements (conditional

Elisa Bertino (Ed.): ECOOP 2000, LNCS 1850, pp.472–491, 2000.

transitions, nested states) resulting from our use case to state machine transformation. We do not know of any other approach to systematically transform use cases via dynamic UML models (part of the software analysis and design and test model) into test suites, which considers the full amount of information provided by a use case document. We thus provide a method which takes all the information from the requirements, incorporates the necessary generalisations made during the analysis phase (represented in a state chart), and enables the verification of the requirements at the end of the software process.

The paper is structured as follows: Section 2 discusses related work on test generation from state machines. Section 3 is concerned with the first part of the transformation, i.e. the generation of a state machine based on a use case document; this section is an abridged version of our discussion in [7]. In section 4 we then focus on how the state machine created from the use cases can be used for test generation. We discuss the relationship between the test generation problem and *STRIPS* planning [5]. We define an algorithm for test suite generation and study an example using the graphplan [2] tool. The appendix provides the *graphplan* input and output for the example discussed in the text.

2 Related Work

There is common agreement in literature, e.g. [11], that use cases should be used for deriving test cases. However, concrete approaches how to do that are rare. Jacobson suggests in [11] three general kinds of tests that can be derived from use cases: (1) tests of the expected flow of events, (2) tests of unusual flows of events and (3) tests of any requirement attached to a use case. Jacobson does not go into the details of how to choose test cases and how to know when you are done.

Binder proposes in [1] to enhance use cases with a few “testability extensions”: the domain of each participating variable, the required input/output relationships among use case variables, the relative frequency of each use case and the sequential dependencies among use cases. He uses these extensions to describe in detail how to develop test cases for use cases and the order in which they should be run.

Unlike use cases state diagrams have long been considered as important for the development of software tests [1][7][13]. One recent discussion can be found in [13]. Therein Marick discusses both simple state machines and state charts as sources for tests. The construction of state charts is described more informally, compared to our method in section 3. The author starts from single scenarios instead of use cases; in this way he has no systematic information on the relationship among the scenarios. Pre- and postconditions of the scenarios as provided by our use case template are not considered. While we use nested states to systematically capture the different levels of sceanarios, Marick [13] uses them informally to group similar states in the state machine. The method for deriving test cases described by Marick takes into account important features of state charts but is described informally, compared to our semi-automatical approach using AI planning methods. The coverage level targeted by Marick's method is to exercise every transition. Our formal approach introduced in section 4 achieves the same coverage level as Marick's manual method.

3 From Use Cases to Dynamic UML Models

Use Cases [11] are a popular formalism for capturing functional requirements and business requirements. Use cases are a good means to communicate with a customer. They are the unit of work in incremental object-oriented software processes like the Rational Unified Process [12]. Furthermore, they are a good basis for systematic testing. In [19] Rumbaugh et al. define a use case as "the specification of sequences of actions, including variant sequences and error sequences that a system, subsystem or class can perform by interacting with outside actors". While the advantages of use cases for requirements engineering are widely accepted, the impact of use cases on software design is less clear. The UML meta model [15] offers three different notations for designing dynamic system/subsystem/class behaviour based on use cases, as discussed in [20]:

- *Activity diagrams* interpret use cases as branching processes. Some authors [21] recommend their use for the formalisation of use cases. However, doing this has considerable disadvantages: First, state diagrams correspond directly to the object-oriented paradigm, whereas activity diagrams model the control flow of a program. Thus they provide the same means for creating a spaghetti control-structure as do flow diagrams. Second, the distinction between normal and abnormal behaviour, which is one of the benefits of use case analysis is lost.
- *Interaction diagrams* (sequence diagrams and collaboration diagrams) can be used to formalise single scenarios contained in use cases. In [18] Rumbaugh proposes to start dynamic modelling with scenarios in text form, formalise them as sequences of events in interaction diagrams and merge them subsequently into state diagrams showing the complete lifecycle of an object. This seems to be a rather unnatural approach for formalising use cases. Since a use case is a hierarchical collection of scenarios, these have to be formalised separately in interaction diagrams and then merged again into a single state diagram. [8][18][19][4]
- *State machines* specify the behaviour of a system/subsystem/class in reaction to events from actors. In contrast to interaction diagrams they visualise multiple scenarios, e.g. the hierarchy of scenarios described by a use case (see section 3.1).

3.1 Use Cases to State Machines

This section briefly describes a transformation from use cases to state machines, which we describe more thoroughly in [7]. State machines and state diagrams have a long history in computer science. Recent versions of UML [15] include an expressive state diagrams concept inspired by Harel's state charts [8][9][10]. Especially the abstraction mechanisms in the UML state machine formalism, i.e. nesting of states and stubs, allow us to map all the important elements of our use case documents to state machines.

Use Case Documents

The following simple example describes, how a library user borrows a book. The structure of the use case was inspired by Alistair Cockburn's use case template [3]. This template describes the scenarios of the use case in a hierarchical fashion. The

Main Success Scenario is the straightforward sequence of steps leading to the achievement of the user's goal without consideration of possible problems. With each step, possible error situation and their resolution can be described in the *Extensions* section. Further, there may be different alternative ways to execute a step (e.g. search a book by title or search by author). These alternatives are described in the *Variations* section. The *Preconditions* section captures constraints, which the state of the world must satisfy before the use case can be executed. These are typically properties of the user (e.g. the user must have an account) or the state of program execution (e.g. the user is logged in). The *Postconditions* section on the other hand describes the conditions, which the use case establishes. Thus, Pre- and Postconditions together define the *contract* of the use case [14]. A step in the use case can refer to another use case being called. In the *Borrow Book* example below, step 8a1) references the *Log in* use case. The referenced use cases are summarised in the *Included Use Cases* section.

Name	Borrow Book
Goal	This use case describes how a library user selects and then borrows a book from the library.
Preconditions	None
Postconditions	The user is registered as the borrower of the book in the library system.
Main Success Scenario	1. The user selects the search function from the main menu. 2. The system displays the search form. 3. The user enters the title of a book (possibly using wild-cards). 4. The library system presents a list of all matching books 5. The user selects a book. 6. The system displays the detail view for this book. 7. The user selects borrow from the menu for this book. 8. The user is already logged in. The system issues a message to the archive that the book is reserved for the user.
Extensions	4a) There are no matches to the query. 4a1) The system returns to the main screen. 8a)The user is not logged in. 8a1)The user logs in as described in Log in.
Variations	3a) The user enters the name of the author. 3b)The user selects the author from an author list. 3b1)The user clicks on "select author". 3b2)The system displays a selection list of all authors. 3b3)The user selects an author from the list.
Included Use Cases	Log in

Name	Log in
Goal	This use case describes how a library user logs into the system to prove his identity.
Preconditions	None
Postconditions	The user is logged in

Name	Borrow Book
Main Success Scenario	1. The user selects log in from the main menu. 2. The system asks the user for his login name. 3. The user enters his login name. 4. The system asks the user for his password. 5. The user enters his password. 6. The system verifies login and password. They are ok. 7. The system logs the user on.
Extensions	6a) The combination of login and password is not ok. 6a1) If the number of retries is not exceeded, repeat the use case from step 2.
Variations	None
Included Use Cases	None

In the following sections, we will describe how all elements of a use case document are mapped to a UML state machine.

Main Success Scenario

Each step in a use case corresponds to a message sent by an actor to the system or vice versa. The interval between two messages sent to the system is an abstract state of the sys-tem. As proposed by Rumbaugh [18], we denote all messages sent by the system as actions of the state. Each message sent by an actor is denoted as an event, causing a transition between two states of the system. The beginning of the use case is modelled by an initial state; the use case ends in a final state of the state machine. As intended in use case analysis, the main success scenario ends with the successful achievement of the goal [3]. Thus, the final state reached after the last step of the use case corresponds to successful completion. The whole state diagram is encapsulated in a super state named after the use case for later reuse - to model relationships to other use cases. **Fig. 1** shows the state diagram corresponding to the *Borrow Book* use case.

Each UML state diagram corresponds to an or an abstract system object, e.g. the Library. The events shown in the state diagram, e.g. *Enter Title* and *Select Search*, are not necessarily methods of a concrete class but events the system understands. These map to statements in a test script for automatic system testing.

Variations

Variations in use cases are usually local alternatives for executing a step in the use case. In a state diagram these can be modelled as multiple paths connecting two states. In the simplest case, a variant can be represented as an additional link between the two states delimiting the corresponding step in the main success scenario. Intermediate states may be needed to model more complex variants. The state chart modifications needed in our example for variants 3a) and 3b) are shown in Fig. 2.

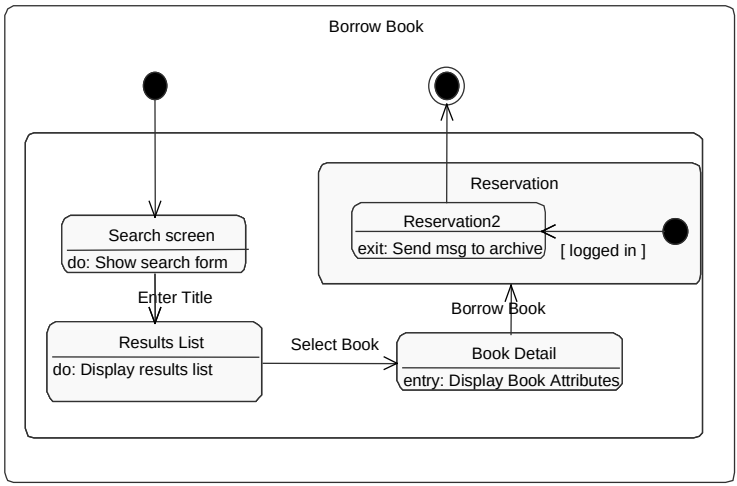

Fig. 1. Basic state diagram capturing the main success scenario.

Extensions

An extension usually describes a backup solution for completing a subgoal in a use case. A special case of this is when a subgoal (step) fails, because a precondition does not hold as in the example's extension 8a). A modular way to handle extensions is to specify them using substates of the state representing the corresponding step in the main success scenario. Extension 4a) is an exception, where we abandon the goal of the use case. No book is found and the library system goes into an error state.

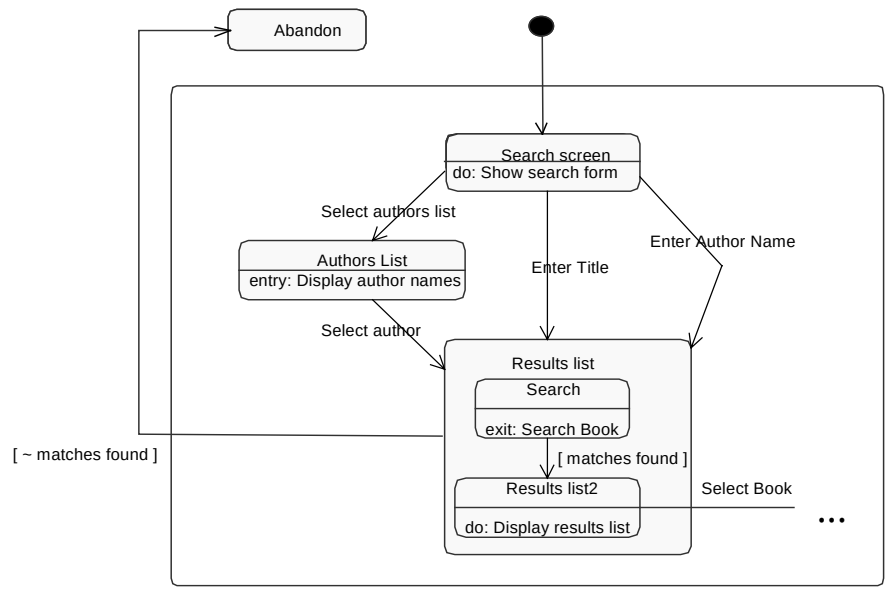

Fig. 2. State diagram with variations and extensions. The Log in state is currently a placeholder.

Preconditions and Postconditions

The Preconditions and Postconditions sections of the use case template allow us to specify the contract of the use case [14]. Preconditions describe verifiable conditions, which must hold before the execution of the use case. We model the preconditions of the use case as constraints on the first state representing the use case. The upper part of **Fig. 3** shows how a precondition on a state can be modelled in UML using a superstate with two substates.

We model the postconditions of a use case as constraints on the last state representing the use case. The lower part of **Fig. 3** shows how a postcondition on a state can be modelled using a superstate with two substates. In contrast to our previous work [7], we now model postconditions of the use case by actions which assure the postcondition of the use case (add *C*). These actions formalise the idea that the use case establishes the postconditions on successful completion. We exploit these action statements during test suite planning, when we match the actions establishing a condition with the preconditions of other use cases or steps requiring this condition.

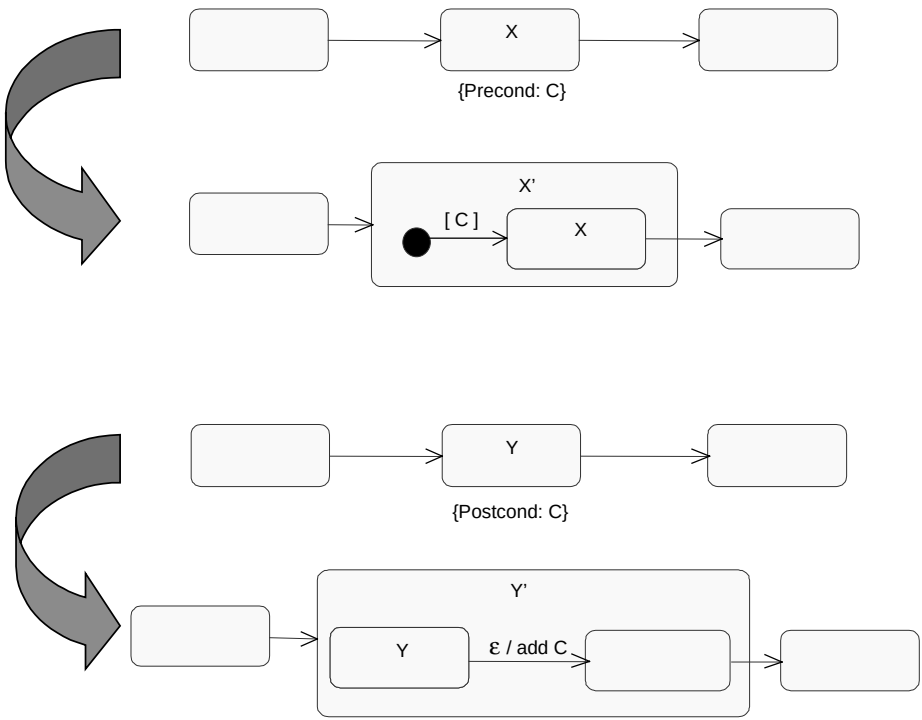

Fig. 3. Modelling Pre- and Postconditions using UML State Diagrams

Subordinate Use Cases

UML defines a mechanism for including one use case as a subfunction in another use case. This information is covered by the subordinate use case section of our template. In our example, the *Borrow Book* use case includes the functionality of the *Log In* use

case. To integrate the *Log In* use case with the *Borrow Book* use case we use two techniques from the UML state diagram notation:

- A submachine reference state allows us to copy the state machine formalising the subordinate use case (*Log in*) into the enclosing use case (*Borrow Book*).
- Stub states allow us to connect the states in the submachine (*Log in*) to the right states in the enclosing machine (*Borrow Book*).

Fig. 4 shows how the use case *Log In* is embedded into *Borrow Book*.

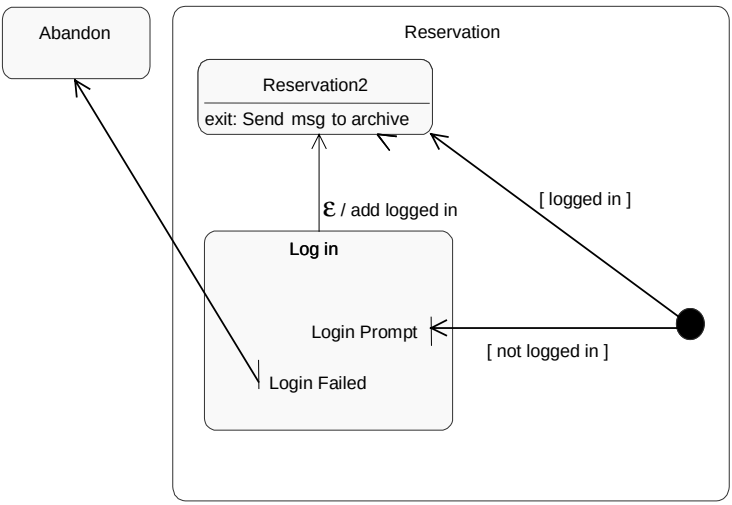

Fig. 4. Connection of Borrow Book and Log in use case.

4 From State Machines to Test Cases

Using state models to derive test cases has been common practice in the software testing world for some time [13]. The final goal of model-based testing is to automate the test case generation from test models as much as possible.

The approach presented in the current chapter takes automation further than previous approaches by interpreting the preconditions and actions of the transitions. Our algorithm generates a set of valid test sequences, where the preconditions of all transitions are established either by previous actions or by properties of the test data.

This is made possible by exploiting AI planning techniques, which allow us to systematically search for paths in the state machine, which satisfy all preconditions of the transitions. In particular, we describe the test generation problem as a STRIPS planning problem [5] and solve it with the graphplan tool [2].

The scope of our method is the generation of test sequences supplemented by constraints on the test data, as far as these can be derived from the information present in the state machine.

4.1 Properties of Considered State Models

One of the properties of our transformation from use cases to state machines is that the resulting state models have conditional transitions. As shown in the example in section 3.1, we introduce an identifier (e.g. *logged in*) for each pre- and postcondition in a use case or use case step. From now on, we will call these conditions *Propositions* and denote the set of all propositions in the state machine by *A*.

The propositions usually encode properties of the session or user on a semantical level. Typical propositions are *connected*, *logged in*, *items paid*, *request pending*, or *unsaved changes*. Note that the state space of a piece of software modelled in our approach consists of multiple dimensions: the state of the execution modelled by states and transitions of the state machine, and the state of the session or user modelled by propositions.

4.2 Structure of a STRIPS Planning Problem

Before we establish the connection between test generation and STRIPS planning, we have to introduce some STRIPS [5] terminology. In the search space of a STRIPS planning problem each state is described by a set of propositions which hold in that state. A set of operators describes the transitions among the states. The planning task is to find a sequence of operators which safely connects the initial state (called Σ) to the final (goal) state (called Ω). The fundamental advantage of the STRIPS language lies in the way the operators are defined.

An operator $\alpha = (Pre, Add, Del)$ is defined by

- A set *Pre* of propositions, called *Prerequisites of* α. These prerequisites must be true in every state to which α is applied.
- A set *Add* of propositions, which are true in every state resulting from the application of α, i.e. *Add* describes the propositions which are established by applying α.
- A set *Del* of propositions, which may no longer hold in a state directly resulting from the application of α. This means, every other proposition $p \in A \setminus Del$ remains true in the state resulting from application of α, if it was true before.

The power of this operator definition is that it only describes which propositions are established or possibly deleted by the application of α. All propositions not mentioned maintain their current values. This definition without the explicit description of what remains constant avoids the so-called *frame problem* [17], from which many other representations suffer. The STRIPS formalism is a widely accepted language for specifying planning problems and supported by many state of the art planning tools.

In examples of this paper, we use only propositional logic in the definitions of the operators, i.e. sets of atoms, which contain no variables. The planning tool we use allows in addition the use of free variables. With this simple extension we can define an operator *show*(*X*), which has *invisible*(*X*) as prerequisite and *visible*(*X*) in the Add-list. Since the planning tool requires the domain of *X* to be finite and defined, this is a safe use of variables, which does not confront us with the problems of first-order logic, since all variables can be instantiated based on the given domain [16].

4.3 Basic Generation of a Planning Problem

Let us first define a planning problem, which yields a test case including a selected state or a selected transition. In section 4.4 we will generalise our results leading to an algorithm for generating a set of test cases covering all states or all transitions. Consider the example shown in **Fig. 5**, which is a simplified version of the example from section 3.1. For easier reference, we have labelled each transition with a name (*t0* through *t9*).

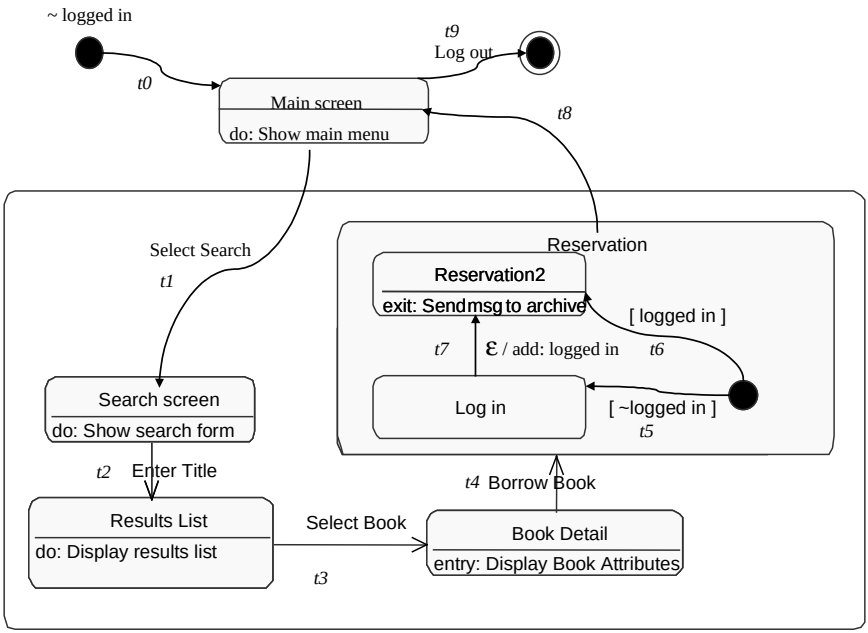

Fig. 5. Simple state machine example for test case generation.

Operator Definition

In the planning problem we represent each transition of the state machine by an operator. To specify the operators we need some additional propositions:

- For each state *s* in the state machine, we introduce two propositions:
  - a proposition *in(s)*, denoting that the current state of execution is *s* and
  - a proposition *log_state(s)*, denoting that the state *s* was the current state at some point in time.
- For each transition *t* in the state machine, we introduce a proposition *log_trans(t)*, denoting that the transition *t* was used at some time.

Now consider the transition *t* shown in **Fig. 6**, which contains all elements of a transition (event, condition, and action). We represent this transition by the operator

$$\alpha := (Pre, Add, Del),$$

where

Pre = {$in(s_0)$, p}
Add = {$in(s_1)$, $log\_state(s_0)$, $log\_trans(t)$, $p1$}
Del = {$in(s_0)$}

This operator definition models both the transitions among the states of the state machine and the value changes of propositions. The prerequisites specify that the operator can only be applied, if the current state is s_0 ($in(s_0)$) and the proposition p is true. The *Add*-list shows that after the application of the operator the state is s_1 ($in(s_1)$). We create log-entries for the facts that we were in state s_0 ($log\_state(s_0)$) and that we used the transition t (log_trans(t)).

The change in the set of propositions is expressed by adding *p1*, because the proposition *p1* is made true by the action *Add: p1*. The *Del*-list shows that the current state is no longer s_0 ($in(s_0)$). The truth value of p and all other propositions not mentioned on the *Add* or *Del*-list is guaranteed to be preserved.

The operator definition is completely analogous for a transition with a *Del*-action. If the action would be *Del: p1* instead of *Add: p1* in **Fig. 6**, we would put *p1*on the *Del*-list instead of the *Add*-list. In case the condition and / or action are missing on a transition, the operator definition is the same, just without the corresponding entries in the *Pre*, *Add*, and *Del*-lists.

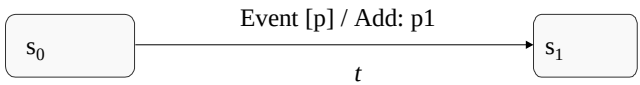

Fig. 6. Transition *t* with Event, Condition, and Action

The following table gives the operator definitions for the state machine from **Fig. 5**. As a convention, we call the initial state of the whole state machine *initial*, and the final state of the whole state machine *final*. Similarly, we denote the initial state within a state *s* by *initial(s)* and the final state within a state *s* by *final(s)*. Further, we use the character ¬ to denote the negation of a proposition, e.g. ¬ *logged in* reads "*not logged in*".

Operator	Pre	Add	Del
α0	in(initial)	log_state(initial), log_trans(t0) in(Main Screen)	in(initial)
α1	in(Main Screen)	log_state(Main Screen) log_trans(t1) in(Search Screen)	in(Main Screen)
α2	in(Search Screen)	log_state(Search Screen) log_trans(t2) in(Results List)	in(Search Screen)
α3	in(Results List)	log_state(Results List) log_trans(t3) in(Book Detail)	in(Result List)
α4	in(Book Detail)	log_state(Book Detail) log_trans(t4) in(initial(Reservation))	in(Book Detail)

Operator	Pre	Add	Del
α5	in(initial(Reservation)) ¬ logged in	log_state(initial(Reservation)) log_trans(t5) in(Log in)	in(initial(Reservation))
α6	in(initial(Reservation)) logged in	log_state(initial(Reservation)) log_trans(t6) in(Reservation2)	in(initial(Reservation))
α8	in(Reservation2)	log_state(Reservation2) log_trans(t8) in(Main Screen)	in(Reservation2)
α9	in(Main Screen)	log_state(Main Screen) log_trans(t9) in(final)	in(Main Screen) logged in

Initial State

Having the operator definitions in place, we have to specify the initial conditions and the goal in order to arrive at a complete planning problem, which can be solved by a planning tool. The initial state Σ of our planning problem specifies that we start the execution of a test in the initial state of the state machine. Moreover, the initial state has to specify for each proposition $p \in A$, if it is true of false in the beginning of the execution. For propositions, which keep track of the program execution, we will always assume that they are false in the initial state. Thus, in our small example, we assume that the proposition *logged in* is false. Based on these considerations, we have

$$\Sigma = \{\neg \textit{ logged in}, \textit{in}(\textit{initial})\}.$$

Goal

We have included *log-state* and *log-trans*-entries in the operator definitions to keep track of the states and transitions covered during the execution of the test case. By using these entries in the formulation of the goal state, we can instruct the planner to create test cases, which include certain states and transitions. As an example, let us derive a test case, which tests the transition *t6*, which leads to the *Reservation2* state, in case the user is already logged in. The corresponding goal statement is

$$\Omega = \{\textit{log-trans}(\textit{t6}), \textit{in}(\textit{final})\}.$$

The presence of the proposition *in*(*final*) makes sure, that the test case ends in the final state of the state machine.

Computation of the Test Case

In Appendix A, we show the graphplan input files corresponding to the planning problem described in the previous sections. The graphplan execution trace shown at the end of Appendix A points out that graphplan generates the desired test sequence in 0.06s. The test sequence consists of 15 steps (borrow two books in sequence, so that the second time the user is already logged in).

This small proof of concept example had the goal to illustrate the basic steps in test sequence generation using a planning tool. In the following section, we will derive a systematic algorithm for generating a set of test sequences with a given coverage level.

4.4 Algorithms for Achieving a given Coverage Level

Properties of the Test Data

Let us now reconsider the meaning of our propositions: Propositions represent the pre- and postconditions of the use cases. In our previous example, the proposition *logged in* represented the fact that the user has already logged into the system. This condition can be achieved through successful completion of the *Log in* use case. In general however, we cannot assume that all conditions are achievable by use cases. They can as well represent properties of the test data / test input. As an example, consider **Fig. 7**: In this modified version of our state machine example, we have added a check whether the selected book is available and expressed it by the proposition *Book available*. Obviously, there exists no transition in the state model which makes *Book available* true. The truth value of *Book available* depends entirely on the title of the book, which the user enters. Thus, we cannot guarantee that the book will be available only by the design of a test sequence, but we have to record that the test input must have this property. We assure this using a two-step approach:

1. We determine all propositions in the state machine which do not occur in an *add* or *del*-action.
2. For each proposition *p* found in step 1, we create two operators:

$$\alpha_p^+ = (Pre, Add, Del), \text{ where } Pre = \emptyset, Add = \{p\}, Del = \emptyset$$
$$\alpha_p^- = (Pre, Add, Del), \text{ where } Pre = \emptyset, Add = \{\neg\, p\}, Del = \emptyset$$

In the example from **Fig. 7**, this procedure adds an operator which makes *Book available* true and one, which makes it false. Before e.g. transition *t8* can be passed, the operator, which makes *Book available* true must be applied. From the plan, our algorithm can see that this pseudo-operator has been applied and derive that the test input has to have the property *Book available*. Thus, in addition to the test sequence, we also formally deduce a set of constraints for the test data (in this case the availability of the book).

Test Coverage

The weakest coverage criterion for state machine testing is to cover every state. The next better coverage level, which is equivalent to branch coverage, is to cover every transition. This is proposed e.g. in [13]. To achieve this coverage level, we construct the test suite iteratively. We start by marking every transition as *not covered*. Then, we generate the first test sequence and mark every transition in the test sequence as *covered*. Then, in a loop, we pick one transition not yet covered and create a test sequence, which includes this transition (and maybe some further previously not covered transitions). We have described how to create such a test sequence including one selected transition in section 4.3. We repeat this procedure until all transitions are covered by at least one test sequence.

A complication of the above mentioned procedure arises in the case of nested state machines. For example, in our state machines from **Fig. 5** and **Fig. 7**, *Log in* is a sub-machine reference state, representing the state machine resulting from the *Log in* use case. There are two ways to handle test coverage for sub-machine reference states.

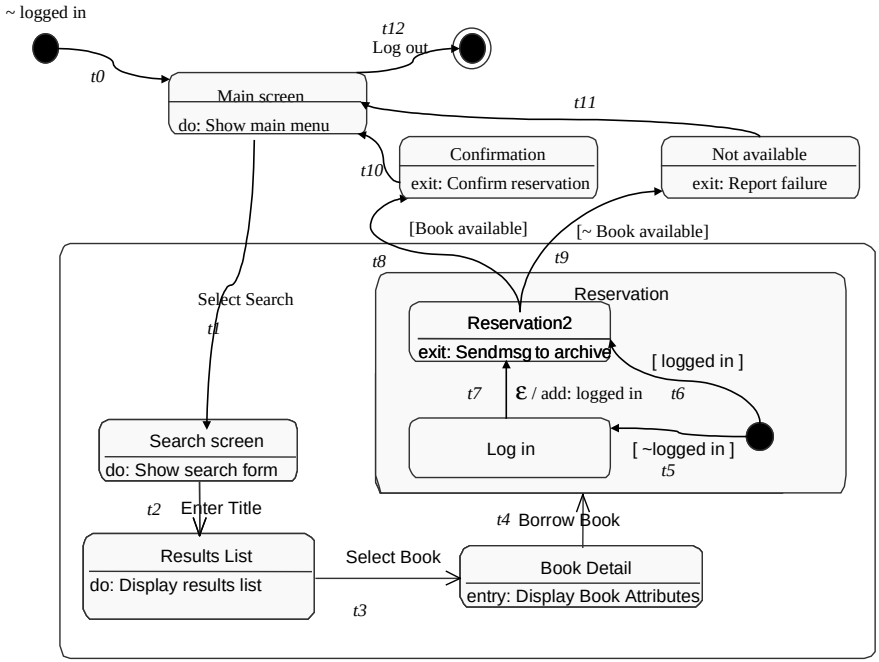

Fig. 7. Modified state machine example. It adds the possibility that the book selected by the library user is not available (e.g. already borrowed by someone else).

- Test separately: If the sub-machine can be tested separately, we can generate a test suite for the sub-machine and ignore coverage for the sub-machine, when designing a test suite for the surrounding state machine. In our example, this may be a sensible choice, because the *Log in* procedure does not depend on the details of the book being borrowed and can thus probably be tested separately.
- Combine test sequences: In the general case, we cannot assume, that the sub-machine is tested separately. For example, if we had a sub use case describing the communication of the library computer with the central database server it would probably be beneficial to test this together with the surrounding borrow book use case, because the communication depends on the book data provided by the user.

We achieve test coverage for the surrounding and included state machines by the following recursive procedure:

- Generate a test suite for the included state machine and record the number *n* of test cases in this suite.
- Generate a test suite for the surrounding state machine with the additional constraint that the sub-machine reference state must occur at least *n* times in the test cases.
- "Folding in": Insert the *n* different test sequences for the sub-machine at the occurrences of the sub-machine reference state in the test sequences for the surrounding state machine.

Algorithm

The considerations of the previous two sections lead to the following algorithm for test suite generation:

Algorithm	Create Test Suite
Input	A state machine *M*.
Output	A list of test sequences and constraints on the test data for each test sequence.
Preprocessing	• Generate the set of operator definitions for *M*. • Collect the set *A* of all propositions occurring in the state machine. • For each proposition $p \in A$: Check if *p* occurs in an action statement of the form *Add*:*p* or *Del*:*p*. If not, generate two new operators for *p*, for recording it as a constraint on the test data: $\alpha_p^+ = (\emptyset, \{test\text{-}data(p), p\}, \emptyset\}$, $\alpha_p^- = (\emptyset, \{test\text{-}data(\neg p), \neg p\}, \emptyset\}$ Create the initial state definition: $\Sigma = \cup_{p \in A} \{\neg p\} \cup \{in(initial)\}$ • Create a list of all transitions and mark each transition as not covered.
Sub-machines	IF sub-machines exist: 1. Create a counter for each state indicating, how often this state must occur in the test suite. 2. Initialise all counters to 1. 3. For each (first occurrence of a) sub-machine reference state: • Call *Create Test Suite* for the sub-machine and record the results. • Set the minimum occurrence counter of the sub-machine reference state to the number of test sequences in the test suite.
Test Generation	WHILE unmarked transitions exist OR an occurrence counter > 0 for any state exists: 1. Select (randomly) an unmarked transition or a state with occurrence counter > 0. 2. Generate the goal definition for a test sequence covering the selected transition or state. 3. Call the planner to obtain the test sequence. 4. Mark all transitions covered by the plan. 5. Decrease the occurrence counters of all states (if they exist) by the number of occurrences of the state in the test sequence. 6. Derive the constraints of the test data from the pseudo operators in the plan. Combine the obtained test sequences with the sequences form the sub-machines. Return the test suite consisting of the test sequences together with the constraints on the test data.

5 Conclusions

In our paper we have discussed a new approach to automatically generate test cases from use cases. We do that in two steps:

1. Formal transformation of a detailed use case description including pre- and postconditions to a UML state model
2. Generation of test cases from the state model

Our formal transformation from use cases was introduced in [7] and was presented in a shortened and slightly revised version here. Our test case generation approach transforms the problem at hand into a planning problem and uses STRIPS - the most widely used AI planning formalism – to derive a test suite for a given state chart. Using this planning technique ensures that the test sequences derived from the state machine are consistent in the sense that the preconditions of all transitions in the sequence are satisfied. This is an improvement over previous methods which generate arbitrary paths and do not consider conditional transitions. First results using the state-of-the-art planner graphplan show the feasibility of our approach. Our method ensures, that a test suite is generated for the system test, which is consistent with requirements and the results of requirements analysis (the UML state model).

With our approach we now have a coherent procedure to derive test cases from use cases in a formal and partly automatic way. Our procedure can however not prevent that some manual additions have to be made:

- The expected system responses have to be added to the test sequence manually to yield complete test cases.
- Although we derive some constraints on the test inputs automatically, the concrete test data still has to be defined manually.

Ongoing and further work concentrates on extensions to our approach: Stronger coverage criteria will be considered. Besides, systematic consideration should be given to other aspects of the specification, e.g. performance and frequency requirements.

References

[1] Robert Binder. *Testing Object-Oriented Systems.* Addison-Wesley, 1999.

[2] A. Blum, M. Furst. *Fast Planning Through Planning Graph Analysis*. Artificial Intelligence, 90:281-300 (1997).

[3] Alistair Cockburn. *Structuring Use Cases with Goals.* Journal of Object-Oriented Programming, Sep/Oct, 1997, pp. 35-40, and Nov/Dec, 1997, pp. 56-62.

[4] Jules Desharnais, Marc Frappier, Ridha Khèdri, and Ali Mili. *Integration of Sequential Scenarios.* IEEE Transaction on Software Engineering, Vol. 24, No. 9, September 1998.

[5] R.E. Fikes, N.J. Nilsson. *STRIPS: a new approach to the application of theorem proving to problem solving.* Artificial Intelligence 2, 1971.

[6] Martin Fowler. *Use and Abuse Cases.* Distributed Computing, April 1998.

[7] Peter Fröhlich, Johannes Link: *Modelling Dynamic Behaviour Based on Use Cases*. Proceedings of Quality Week Europe, Brussels, November 1999.

[8] D. Harel. *Statecharts: A Visual Formalism for Complex Systems.* Science of Computer Programming, Vol. 8, 1987.

[9] D. Harel. *On Visual Formalisms*. Communications of the ACM, Vol. 31, No. 5, Pages 514-531, May 1988.

[10] D. Harel, Michael Politi. *Modeling Reactive Systems With Statecharts: The STATEMATE Approach*. McGraw-Hill, New York, N.Y., 1998.

[11] Ivar Jacobson, Magnus Christerson, Patrick Jonsson, Gunnar Övergaard. *Object-Oriented Software Engineering: A Use Case Driven Approach.* Addison-Wesley, Wokingham, England, 1992.

[12] Ivar Jacobson, Grady Booch, James Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, 1999.

[13] Brian Marick. *The Craft of Software Testing: Subsystem Testing Including Object-Based and Object-Oriented Testing.* Prentice Hall, 1995.

[14] Bertrand Meyer. *Object-Oriented Software Construction.* Prentice Hall, 1997.

[15] Object Management Group. *Unified Modeling Language Specification.* Framingham, Mass., 1998.

[16] R. Reiter. Equality and Domain Closure in First Order Databases. Journal of the ACM, 32:57–97, 1987.

[17] R. Reiter. The frame problem in the situation calculus: A simple solution (sometimes) and a completeness result for goal regression. In: V. Lifshitz (Ed.), *Artificial Intelligence and mathematical theory of computation: Papers in honor of John McCarthy*. Boston: Academic Press, 1991.

[18] James Rumbaugh, Michel Blaha, William Premerlani, Frederick Eddy, William Lorensen. *Object-Oriented Modeling and Design*. Prentice Hall, Englewood Cliffs, N.J., 1991.

[19] James Rumbaugh, Ivar Jacobson, Grady Booch. *The Unified Modeling Language Reference Manual*. Addison-Wesley, Reading, Mass., 1999.

[20] Russell R. Hurlbut. *The Three R's of Use Case Formalisms: Realization, Refinement, and Reification.*Technical Report XPT-TR-97-06, Expertech, Ltd, 1997.

[21] Geri Schneider, Jason P. Winters. *Applying Use Cases: A Practical Guide.* Addison-Wesley, 1998.

Appendix A: Graphplan Operator Definition File

```
(operator alpha0
        (params)
        (preconds (in initial))
        (effects (del in initial)
                 (log-state initial) (log-trans t0)
                 (in main-screen)))

(operator alpha1
        (params)
        (preconds (in main-screen))
        (effects (del in main-screen)
                (log-state main-screen)
                (log-trans t1)
                (in search-screen)))

(operator alpha2
        (params)
        (preconds (in search-screen))
        (effects (del in search-screen)
                (log-state search-screen) (log-trans t2)
                (in results-list)))

(operator alpha3
        (params)
        (preconds (in results-list))
        (effects (del in results-list)
                (log-state results-list) (log-trans t3)
                (in book-detail)))

(operator alpha4
        (params)
        (preconds (in book-detail))
        (effects (del in book-detail)
                (log-state book-detail) (log-trans t4)
                (in initial-reservation)))
(operator alpha5
        (params)
        (preconds (in initial-reservation) (not-logged-in))
        (effects (del in initial-reservation)
                (log-state initial-reservation) (log-trans t5)
                (in log-in)))

(operator alpha6
        (params)
        (preconds (in initial-reservation) (logged-in))
        (effects (del in initial-reservation)
                (log-state initial-reservation) (log-trans t6)
                (in reservation2)))

(operator alpha7
        (params)
```

```
        (precond (in log-in))
        (effects (del in log-in) (del not-logged-in)
                 (log-state log-in) (log-trans t7)
                 (in reservation2)
                 (logged-in)))

(operator alpha8
        (params)
        (precond (in reservation2))
        (effects (del in reservation2)
                 (log-state reservation2) (log-trans t8)
                 (in main-screen)))

(operator alpha9
        (params)
        (precond (in main-screen))
        (effects (del in main-screen) (del logged-in)
                 (log-state main-screen) (log-trans t9)
                 (in final) (not-logged-in)))
```

Initial State and Goal:

```
(preconds
      (not-logged-in)
      (in initial))

(effects
      (log-trans t6)
      (in final))
```

Execution trace of graphplan:

```
alpha0
alpha1
alpha2
alpha3
alpha4
alpha5
alpha6
alpha7
alpha8
alpha9
facts loaded.
time: 1, 5 facts and 3 exclusive pairs.
time: 2, 10 facts and 17 exclusive pairs.
time: 3, 13 facts and 32 exclusive pairs.
time: 4, 16 facts and 50 exclusive pairs.
time: 5, 19 facts and 71 exclusive pairs.
time: 6, 22 facts and 95 exclusive pairs.
time: 7, 26 facts and 135 exclusive pairs.
time: 8, 28 facts and 142 exclusive pairs.
time: 9, 28 facts and 99 exclusive pairs.
time: 10, 28 facts and 88 exclusive pairs.
```

```
time: 11, 28 facts and 79 exclusive pairs.
time: 12, 28 facts and 72 exclusive pairs.
time: 13, 29 facts and 76 exclusive pairs.
time: 14, 29 facts and 75 exclusive pairs.
time: 15, 29 facts and 71 exclusive pairs.
Goals first reachable in 15 steps.
765 nodes created.
goals at time 16:
  log-trans_t6 in_final
1 alpha0
2 alpha1
3 alpha2
4 alpha3
5 alpha4
6 alpha5
7 alpha7
8 alpha8
9 alpha1
10 alpha2
11 alpha3
12 alpha4
13 alpha6
14 alpha8
15 alpha9
0 entries in hash table,
14 total set-creation steps (entries + hits + plan length - 1).
15 actions tried
  0.06 secs
```