

Automated Transition from Use Cases to UML State Machines to Support State-based Testing

Tao Yue¹, Shaukat Ali^{1,2}, Lionel Briand^{1,2}

¹Simula Research Laboratory
P.O. Box 134, 1325, Lysaker, Norway

²Department of Informatics, University of Oslo, Oslo, Norway
{tao, shaukat, briand}@simula.no

Abstract. Use cases are commonly used to structure and document requirements while UML state machine diagrams often describe the behavior of a system and serve as a basis to automate test case generation in many model-based testing (MBT) tools. Therefore, automated support for the transition from use cases to state machines would provide significant, practical help for testing system requirements. Additionally, traceability could be established through automated transformations, which could then be used for instance to link requirements to design decisions and test cases, and assess the impact of requirements changes. In this paper, we propose an approach to automatically generate state machine diagrams from use cases while establishing traceability links. Our approach is implemented in a tool, which we used to perform three case studies, including an industrial case study. The results show that high quality state machine diagrams can be generated, which can be manually refined at reasonable cost to support MBT. Automatically generated state machines showed to largely conform to the actual system behavior as evaluated by a domain expert.

Keywords: Use Case Modeling; UML; State Machine; Model-Based Testing (MBT); State-based Testing; Transformation; Natural Language Processing.

1 Introduction

In the last decade, model-based testing (MBT) has attracted much attention in both industry and academia. This can be seen from a large number of MBT tools which have been produced in the recent years [1]. In addition, MBT test strategies have been developed, which have shown to be highly effective [2]. MBT however relies on complete and precise models for executable test case generation. Developing such models has always been a challenge, especially for large-scale industrial systems, and entails a thorough domain understanding and solid modeling expertise. Oftentimes, developing such models is difficult for Software Quality Assurance teams as they are often not sufficiently acquainted with modeling. On the other hand, these teams are comparatively much more familiar with writing textual use cases and know well the application domain. This paper is an attempt towards assisting the development of high-level (system-level) models (state machines in the current paper) from use case models (UCMods), which are subsequently refined such that executable test cases can be generated using existing MBT tools.

The original motivation of this work is to support test automation for a video conferencing system (VCS) at Cisco Norway [3]. Since video conferencing systems at Cisco exhibits state-based behavior, it is natural to provide support for state-based testing and therefore the behavior of the system is captured using UML state machines. Besides, state machines is the most commonly used notation for model-based test case generation and particularly so in control and communication systems. However, the construction of such state machines is manual, very expensive and error-prone. In this paper, we provide support for automated transformation from UCMods to system-level UML state machines. We targeted system-level state machines since our focus was on system testing to address the needs of our industry partner; however lower level state machines may also be generated provided that more detailed use cases are provided as the input for the transformation.

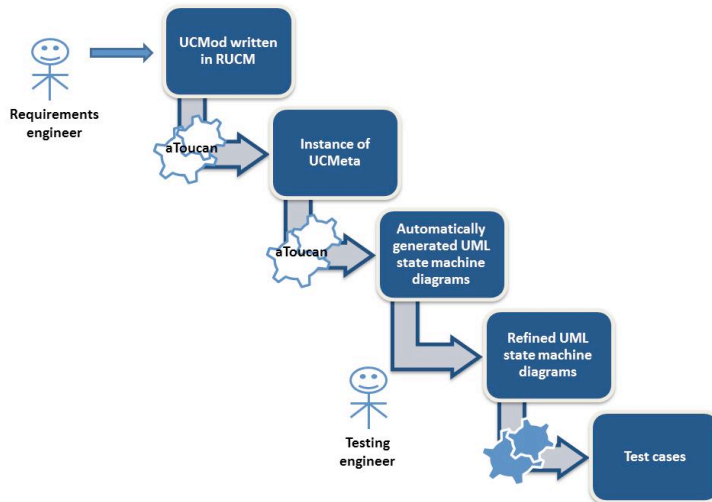


Figure 1 Roadmap from requirements to test cases

Automated transformation from use cases to UML state machines would also enable automated traceability from requirements to state machine diagrams. Traceability is important during software development since it allows engineers to understand the connections between various artifacts of a software system. Traceability is also mandated by numerous standards (e.g., IEEE Std. 830-1998 [4]) to support, for example, change impact analysis or safety verification [5].

The basis of our approach is a use case modeling approach, named RUCM [6], which relies on a use case template and a set of restriction rules for textual Use Case Specifications (UCSs) to reduce the imprecision and incompleteness inherent to UCSs. We have conducted a controlled experiment with human subjects [6] to evaluate RUCM and results indicate that RUCM, though it enforces a template and restriction rules, has enough expressive power, is easy to use, and helps improve the understandability of use cases.

The current work is part of the aToucan approach and tool [7], which aims to transform a UCMod produced with RUCM into a UML analysis model (class and sequence diagrams). aToucan involves three steps. As shown in Figure 1, first,

requirements engineers manually define use cases by following RUCM [6]. Second, aToucan reads these textual UCSs to identify Part-Of-Speech (POS) and grammatical relation dependencies of sentences, and then records that information into an instance of UCMeta (our intermediate metamodel) (Section 3.3). The third step is to transform the instance of UCMeta into a UML state machine. During these transformations, aToucan establishes traceability links between the UCMOD and the generated UML state machine diagrams. These state machines are then refined by test engineers such that executable test cases can be generated [8]. In this work, we focus on the transformation from an instance of UCMeta to a UML state machine diagram.

Two initial case studies were first performed to evaluate the state machine diagrams generated by aToucan. Results show that syntactically correct and mostly complete state machine diagrams were generated. In addition, we also automatically generated a system-level UML state machine in one industrial case study at Cisco Norway. The generated state machine was evaluated by a domain expert in Cisco Norway, who assessed it to mostly conform to the existing, manually developed UML state machine. The latter case study also showed that the refining the generated state machine to get it ready for test generation took less than one hour.

The rest of the paper is organized as follows. The related work is presented in Section 2. In Section 3, we briefly discuss RUCM, UCMeta and the running example being used to exemplify the transformation. The transformation approach is discussed in Section 4, followed by tool support (Section 5). The evaluation, discussion and future work is given in Section 6. Section 7 concludes the paper.

2 Related Work

We conducted a systematic literature review [9] on transformations of textual requirements into analysis models, represented as class, sequence, state machine, and activity diagrams. A carefully designed paper selection procedure in scientific journals and conferences from 1996 to 2008 and Software Engineering textbooks identified 20 primary studies (16 approaches). The method proposed here is based on the results of this review, with a focus on automatically deriving state machine diagrams from UCMODs.

A series of methods is proposed in [10] to precisely capture requirements and then manually transform requirements into a conceptual model composed of object models (e.g., class diagrams), dynamic models (i.e., state machines and sequence diagrams), and functional diagrams. The approach does not purport to provide a solution for transforming requirements into analysis models. Instead, it proposes a set of techniques for users to precisely specify requirements and conceptual models, and also proposes a process to guide the users in deriving the conceptual models from the requirements. No transformation method is reported in the paper.

Somé [11] proposes an approach to generate finite state machines from use cases in restricted Natural Language (NL). The approach requires the existence of a domain model. The domain model serves two purposes: a lexicon for the NL analysis of use cases, and the structural basis of the state transition graphs being generated. The domain model acts as the lexicon for NL analysis of the use cases, because the model elements of the domain model are used to document the use cases. For example, actors of the use cases refer to the classes of the domain model. Interactions between

the system and the actors are defined as one type of use case operations (also including branching statements, use case inclusion statements) which correspond to class operations in the domain model. One can see that a great deal of user effort is needed to obtain a domain model containing classes, associations, and operations, which are indispensable for generating state machines. An algorithm is described in the paper to explain how to automatically transform the use cases plus the domain model into state machines. A working example is used to explain the approach. No case study is presented to evaluate the approach.

Glinz [12] proposes a manual approach to combine a statechart-based structure of the set of scenarios with a structured textual representation of single scenarios to improve requirements quality. Whittle and Schumann proposed an approach [13] to automatically generate UML state machines from UML sequence diagrams.

In summary, none of the existing approaches is able to fully and automatically generate state machine diagrams from a UCMOD while establishing traceability links, which is exactly what we are proposing in the paper.

3 Background

In this section, we briefly review the use case modeling approach RUCM (Section 3.2) and the intermediate model (UCMeta) (Section 3.3) of our transformations (Section 4). The detailed description of RUCM and UCMeta are provided in [6] and [7], respectively. A running example will be presented in Section 3.1 to exemplify RUCM, UCMeta and the transformations.

3.1 Running Example

The running example is a simplified subsystem (called Saturn) of a communication system (video conferencing system) developed by Tandberg [3] (now part of Cisco Norway), which is a leading global provider of telepresence, high-definition video conferencing and mobile video products and services. This subsystem is the industrial case study used to evaluate this work (Section 6).

The core functionality of a typical video conferencing system in Tandberg is sending and receiving multimedia streams. The use case diagram capturing the main functionalities of the simplified subsystem Saturn is given in Figure 2. Saturn deals with establishing video conferencing calls, disconnecting calls, and starting/stopping presentation. It can also receive requests for establishing calls, disconnecting calls, and starting/stopping presentation from other video conferencing systems (Endpoints) participating in a videoconference. The endpoints communicating with Saturn are modeled as secondary actors in the use case diagram in Figure 2.

3.2 RUCM

RUCM encompasses a use case template and 26 well-defined restriction rules [6]. Rules are classified into two groups: restrictions on the use of Natural Language (NL), and rules enforcing the use of specific keywords for specifying control structures. The goal of RUCM is to reduce ambiguity and facilitate automated analysis. Two controlled experiments evaluated RUCM in terms of its ease of application and the quality of the analysis models derived by trained individuals [6,

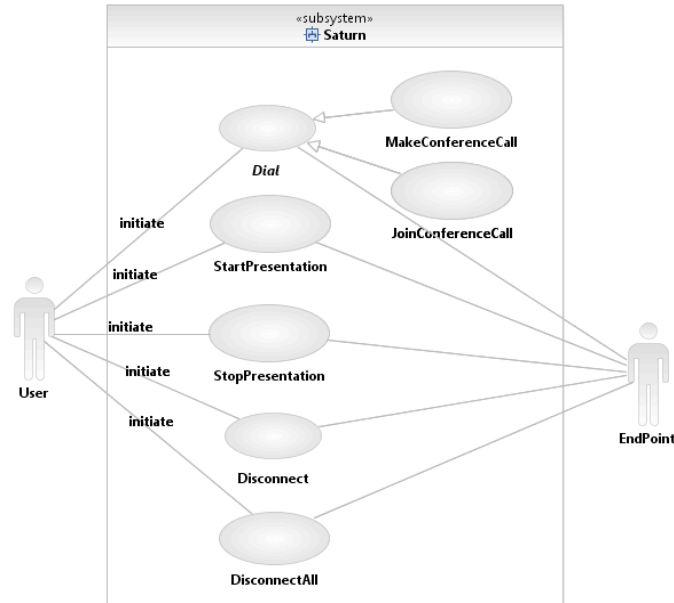


Figure 2 Use case diagram of the subsystem of Saturn

14]. Results showed that RUCM is overall easy to apply and that it results in significant improvements over the use of a standard use case template (without restrictions to the use of NL), in terms of the quality of derived class and sequence diagrams. Below we discuss the features of RUCM that are particularly helpful to generate state machine diagrams. An example of UCS documented with RUCM of use case *Disconnect* (Figure 2) is presented in Table 1. The complete list of UCSs of the use cases in Figure 2 is provided in Appendix A for reference.

A UCS has one basic flow and can have one or more alternative flows (first column in Table 1). An alternative flow always depends on a condition occurring in a specific step in a flow of reference, referred to as *reference flow*, which is either the basic flow or an alternative flow itself. We classify alternative flows into three types: A *specific alternative flow* refers to a specific step in the reference flow; A *bounded alternative flow* refers to more than one step in the reference flow—consecutive steps or not; A *global alternative flow* (called *general alternative flow* in [15]) refers to any step in the reference flow. For example, as shown in Table 1, use case *Disconnect* has one basic flow and two specific alternative flows, which branch from Step 1 and Step 5 of the basic flow, respectively. For specific and bounded alternative flows, a RFS (Reference Flow Step) section specifies one or more (reference flow) step numbers (e.g., *RFS Basic flow 2*). Whether and where the flow merges back to the reference flow or terminates the use case must be specified as the last step of the alternative flow. Branching condition, merging and termination are specified by following restriction rules that impose the use of specific keywords (see below). Each alternative flow and the basic flow must have a postcondition.

RUCM defines a set of keywords to specify conditional logic sentences (IF-THEN-ELSE-ELSEIF-ENDIF), concurrency sentences (MEANWHILE), condition checking

sentences (VALIDATES THAT), and iteration sentences (DO-UNTIL). An alternative flow ends either with ABORT or RESUME STEP, which means that the last step of the alternative flow should clearly specify whether the flow returns back to the reference flow and where (using keywords RESUME STEP followed by a returning step number) or terminates (using keyword ABORT).

Table 1 Use case Disconnect

<i>Use Case Name</i>	Disconnect		
<i>Brief Description</i>	User disconnects an Endpoint participating in a conference call.		
<i>Precondition</i>	The system is in a conference call.		
<i>Primary Actor</i>	User	<i>Secondary Actors</i>	EndPoint
<i>Dependency</i>	INCLUDE USE CASE StopPresentation	<i>Generalization</i>	None
<i>Basic flow steps</i>	1) User sends a message to the system to disconnect an Endpoint. 2) The system VALIDATES THAT Endpoint to be disconnected is in the conference call. 3) The system sends a disconnection notification to Endpoint. 4) Endpoint sends an acknowledgement message back to the system. 5) The system VALIDATES THAT The conference call has only one EndPoint. 6) The system disconnects Endpoint. Postcondition: The system is idle.		
<i>Specific Alt. Flow (RFS Basic flow 2)</i>	1) The system sends a failure message to User. 2) ABORT. Postcondition: The system is in a conference call.		
<i>Specific Alt. Flow (RFS Basic flow 5)</i>	1) The system disconnects Endpoint. 2) ABORT Postcondition: The system is in a conference call.		

3.3 UCMeta

UCMeta is the intermediate model in aToucan [7], used to bridge the gap between a textual UCMOD and a UML analysis model (class, sequence, activity, and state machine diagrams). As a result, we have two transformations: from the textual UCMOD to the intermediate model, and from the intermediate model to the analysis model (Figure 1). Metamodel UCMeta complies with the restrictions and use case template of RUCM. UCMeta is currently composed of 108 metaclasses and is expected to evolve over time. The detailed description of UCMeta is given in [7].

UCMeta is hierarchical and contains five packages: UML::UseCases, UCSTemplate, SentencePatterns, SentenceSemantics, and SentenceStructure. UML::UseCases is a package of UML 2 superstructure [16], which defines the key concepts used for modeling use cases such as actors and use cases. Package UCSTemplate models the concepts of the use case template of RUCM: those concepts model the structure that one can observe in Table 1. SentencePatterns is a package describing different types of sentence patterns, which uniquely specify the grammatical structure of simple sentences, e.g., SVDO (subject-verb-direct object) (Table 1, Basic flow, step 6). SentenceSemantics is a package modeling the classification of sentences from the aspect of their semantic functions in a UCMOD. Each sentence in a UCS can either be a ConditionSentence or an ActionSentence. Package SentenceStructure takes care of NL concepts in sentences such as subject or Noun Phrase (NP). Package UCSTemplate is mostly related to the generation of state machine diagrams and therefore it is the only package discussed below due to space limitation.

Package UCSTemplate not only models the concepts of the use case template but also specifies three kinds of sentences: SimpleSentences, ComplexSentences, and SpecialSentences. In linguistics, a SimpleSentence has one independent clause and no dependent clauses [17]: one Subject and one Predicate. UCMeta has four types

of `ComplexSentences`: `ConditionCheckSentence`, `ConditionalSentence`, `IterativeSentence`, and `ParallelSentence`, which correspond to four keywords that are specified in RUCM (Section 3.1) to model conditions (IF-THEN-ELSE-ELSEIF-THEN-ENDIF), iterations (DO-UNTIL), concurrency (MEANWHILE), and validations (VALIDATES THAT) in UCS sentences. UCMeta also has four types of special sentences to specify how flows in a use case or between use cases relate to one another. They correspond to the keywords RESUME STEP, ABORT, INCLUDE USE CASE, EXTENDED BY USE CASE, and RFS (Reference Flow Step).

4 Approach

Recall that our objective is to automatically transform a textual UCMOD expressed using RUCM into UML state machine diagrams, while establishing traceability links. Notice that the transformation from the textual UCMOD to the instance of UCMeta is not discussed in this paper, but provided in [7] for reference. In this section, we however only focus on the transformation from instances of UCMeta to UML state machine diagrams. We present detailed transformation rules in Section 4.1. The steps required transforming automatically-generated state machine diagrams into the state machines that can be used for automated test case generation in Section 4.2.

4.1 Transformation Rules

The transformation from an instance of UCMeta to a state machine diagram involves 12 rules, which are summarized in Table 2. Subscripts on rule numbers (Column 1, Table 2) indicate the type of the rule: "c" and "a" denote composite and atomic rules, respectively; a composite rule is decomposed, whereas an atomic rule is not.

Rule 1 generates an instance of UML `StateMachine` for a UCMOD, which can then be visualized as a state machine diagram. Rules 2-4 generate the initial state (an instance of `Pseudostate`), the start state (an instance of `State`), and the transition (an instance of `Transition`) from the initial state to the start state, respectively. The reason of requiring the start state is that for a state machine, it is typically required to transit from the start state to more than one state through more than one transition. However, an initial state of UML can have at most one outgoing transition [16].

Composite rule 5 invokes atomic rules 5.1-5.4 to process each use case of the UCMOD. Notice that we generate a single state machine for the whole UCMOD. The generated state machine is a system-level state machine where we model the states of the system as a whole and the system-level calls (e.g., APIs) as transitions. In contrast, in class-level state machines, states are modeled based on class instance variables and transitions have triggers with, for instance, call events as method calls. System-level state machines are at a higher level of abstraction than class-level state machines and are more suitable for system-level testing - the main motivation of this paper. To automatically generate system-level state machines, we utilize the information mostly contained in the precondition and postconditions of all the use cases of a UCMOD to generate states. We generate an instance of `State` for each sentence of the precondition and the postconditions, but making sure that no duplicate states are generated. Since the precondition of a use case indicates what must happen before the use case can start and the postconditions of the use case specify what must

be satisfied after the use case completes, we define rule 5.3 to generate transitions between the states derived from the precondition and the states derived from the postcondition of the basic flow.

Table 2 Summary of transformation rules

Rule #	Description
1 _a	Generate an instance of <i>StateMachine</i> for the use case model.
2 _a	Generate the initial state (instance of <i>Pseudostate</i> with <i>PseudostateKind</i> = <i>initial</i>) for the state machine.
3 _a	Generate an instance of <i>State</i> , named as 'start', representing the start state of the state machine.
4 _a	Generate an instance of <i>Transition</i> . Its trigger is named as 'construct'. This transition connects the initial state to the start state.
5 _c	Invoke rules 5.1-5.4 to process each use case of the use case model.
5.1 _a	Generate an instance of <i>State</i> for each sentence of the precondition of the use case, as long as such a state has not been generated, which is possible because two use cases might have the same preconditions.
5.2 _a	Generate an instance of <i>State</i> for each sentence of the postcondition of the basic flow of the use case, as long as such a state has not been generated.
5.3 _a	Connect the states corresponding to the precondition to the states representing the postcondition of the basic flow of the use case with the transition whose trigger is the name of use case. See Figure 3 for the corresponding Kermeta code.
5.4 _c	Process the postcondition of each alternative flow of the use case.
5.4.1 _a	Generate an instance of <i>State</i> for each sentence of the postcondition of each alternative flow.
5.4.2 _a	Connect the states corresponding to the precondition of the basic flow to the states corresponding to the postconditions of the alternative flows with the transition whose trigger is the name of use case. See Figure 4 for how guard conditions are determined.
6 _a	Connect the precondition of the including use case to its postcondition through a transition. Its trigger is the including use case. Its effect is the included use case, and the guard is the conjunction of all the conditions of the condition sentences of the previous steps of the step containing the INCLUDE USE CASE sentence.

Figure 3 presents an excerpt of Kermeta [18] code implementing rule 5.3 to generate an instance of *Transition* (*pre2post_transition* : *uml::Transition*). The first section of the excerpt is to show how we determine the guard condition of a transition (*SI*). As shown in Figure 3, if the steps of the basic flow of a use case contains *ConditionalSentenceS* (*step.asType(ConditionalSentence)* and/or *ConditionCheckSentenceS* (*step.isInstanceOf(ConditionCheckSentence)*), then the guard condition of the generated transition should be the conjunction of the conditions of these sentences. For example, as shown in the state machine diagram generated for the subsystem of Saturn in Appendix B, the transition from state *The system is in a conference call* (the precondition of use case *Disconnect*, as shown in Table 1) to state *The system is idle* (the postcondition of the basic flow of *Disconnect*) with trigger *Disconnect* and effect *The system disconnect Endpoint*, has the guard condition: *Endpoint to be disconnected is in the conference call. AND The conference call has only one Endpoint*. The guard condition is the conjunction of the two condition checking sentences (i.e., *VALIDATES THAT* sentences) of steps 2 and 5 of the basic flow (Table 1). Thanks to RUCM and UCMeta, the generation of precise guard conditions becomes feasible and easier. This is because RUCM defines a set of keywords (e.g., *VALIDATES*

THAT) and UCMeta formalizes them as metaclasses (e.g., ConditionCheckSentence). Recall that the transformation from a UCMOD expressed with RUCM to an instance of UCMeta is automated and presented in [7].

```

precondition_states.each{precondition_state|
  postcondition_states.each{postcondition_state|
    var pre2post_transition : uml::Transition
    var guard_string : uml::String init uml::String.new
    //S1: Determine the guard condition
    basicflow.steps.each{step|
      if step.isInstanceOf(ConditionalSentence) and
        guard_string := guard_string + " AND " + step.
          asType(ConditionalSentence).IFcondition.description
      end
      if step.isInstanceOf(ConditionCheckSentence) then
        guard_string := guard_string + " AND " + step.
          asType(ConditionCheckSentence).condition.description
      end
    }
    //S2: Generate the guard condition of the transition
    var guard_expression : uml::OpaqueExpression init uml::OpaqueExpression.new
    guard_expression.body.add(guard_string)
    var guard : uml::Constraint init uml::Constraint.new
    guard.specification := guard_expression
    //S3: Generate the trigger of the transition
    var trigger : uml::Trigger init uml::Trigger.new
    trigger.name := uc.name
    //S4: Generate the effect of the transition
    var effect : uml::OpaqueBehavior init uml::OpaqueBehavior.new
    var lastStep : String init String.new
    lastStep := basicflow.steps.at(basicflow.steps.size()-1).
      asType(Sentence).description
    effect.body.add(lastStep)
    effect.name := lastStep
    //S5: Generate the transition
    pre2post_transition := createTransition(statemachine, region,
      precondition_state, postcondition_state, uml::TransitionKind.external,
      guard, trigger, effect)
  }
}

```

Figure 3 Excerpt of Kermeta code for implementing rule 5.3

The second section of the excerpt (S2) in Figure 3 shows how the guard condition is instantiated as an instance of `uml::Constraint`. The third section of the excerpt (S3) generates the trigger of the transition. Its name is assigned as the name of the use case, which is reasonable because the use case triggers the transition from its precondition to the postcondition of its basic flow. The fourth section of the excerpt (S4) generates the effect of the transition. The effect is determined by the last step of the basic flow. One may argue that the steps in the basic flow (except the condition sentences) all together should be considered as the effect. However, according to our experience in developing the tool, we noticed that in most cases, the last step is sufficient to indicate the effect of the transition. In the future, when more case studies are performed, this transformation is expected to be refined. The last fragment (S5) of the excerpt invokes an operation to generate the transition.

Rule 5.4 processes the postconditions of each alternative flow of the use case. Notice that RUCM enforces that each flow of events (both basic flow and alternative flows) of a UCS contains its own postcondition (Section 3.2). This characteristic of

RUCM makes the transformation (rule 5.4) systematic. For a traditional use case template without enforcing this RUCM characteristic, the UCSs expressed with it would have a single postcondition for a use case. Therefore unavoidably the postcondition will combine all the conditions of the basic flow and the alternative flows, hence resulting in a single state corresponding for the postcondition of the use case. Such a state is imprecise because it encapsulates all the interesting states that a system can transit to while executing a particular use case. It is desirable (both for testing and in general) to have separate states and separate transitions with different conditions to handle alternative flows. This is exactly what our transformation does with the help of RUCM.

The reason of handling the postconditions of the alternative flows separately from the transformation rules of handling the basic flow is that guard conditions are processed in a different way. Rule 5.4 further invokes rules 5.4.1 and 5.4.2 to generate states for the postcondition of each alternative flow and connect these states to the already generated states corresponding to the precondition of the basic flow through transitions, respectively. An excerpt of Kermeta code of this rule is provided in Figure 4, which is mainly used to explain how the guard conditions of the transitions are determined. As one can see for the code, the guard condition of a transition from a state generated for the precondition of the use case and a state generated for the postcondition of an alternative flow is determined by negating the condition sentence (either a `ConditionalSentence` or a `ConditionCheckSentence`) of the basic flow that the alternative flow branches from (indicated using keyword `RFS` in the UCS of the use case) (Section 3.2). For example, as shown in the state machine diagram in Appendix B, the transition of state `The system is in a conference call` (representing the precondition of use case *Disconnect*, as shown in Table 1) to itself (representing the postcondition of the first specific alternative flow of *Disconnect*) with trigger `Disconnect` and effect `The system sends a failure message to User`, has the guard condition: `NOT Endpoint to be disconnected is in the conference call`.

```

var guard_string1 : uml::String init uml::String.new
altflow.postCondition.postConditionSentences.each{sen|
  var alt_postcondition_state : uml::State
  alt_postcondition_state := createState(statemachine, region, sen.description)
  precondition_states.each{source|
    altflow.bfs.each{bfs|
      if bfs.isInstanceOf(ConditionalSentence) then
        guard_string1 := "NOT "
        bfs.asType(ConditionalSentence).IFcondition.description
      end
      if bfs.isInstanceOf(ConditionCheckSentence) then
        guard_string1 := "NOT " +
          bfs.asType(ConditionCheckSentence).condition.description
      end
    }
  }
}

```

Figure 4 Excerpt of Kermeta code for implementing rule 5.4.2

Rule 6 processes each include relationship of the UCMOD. For each include relationship, transitions are generated to link the precondition of the including use case to its postcondition, with the trigger named as the name of the including use case, the effect named as the name of the included use case, and the guard being as the

conjunction of all the conditions of the condition sentences of the previous steps of the step INCLUDE USE CASE. As shown in Figure 5, use case *ReturnVideo* includes use case *ReadBarCode* at the basic flow step 2. Transitions are created to connect the precondition of use case *ReturnVideo* to its postconditions. For example, a transition is generated to connect the precondition of *ReturnVideo* ('Employee is authenticated') to its basic flow postcondition (sentence 'A video copy has been returned.' and sentence 'The video copy is available for rent.'), with trigger '*ReadBarCode*', effect '*ReturnVideo*'. Since there is no condition sentence before the basic flow step 2, no guard condition should be generated. Due to space limitation, we cannot provide the complete UCS in the paper.

In terms of Extend relationships between use cases, we don't need specific rules to handle them. As shown in Figure 5, use case *VideoOverdue* extends use case *ReturnVideo* in the alternative flow 1, when 'The system VALIDATES THAT the video copy is not overdue' (Basic flow step 5) is not true. When we follow rules 5.3 and 5.4.2, transitions are generated to connect the precondition of *VideoOverdue* and its postconditions of both the basic and alternative flows. Notice that the precondition of *VideoOverdue* is the combination of the precondition of *ReturnVideo* and the negative condition of the basic flow step 5. Therefore, there is no need to create transitions from the precondition of *ReturnVideo* to the postconditions of *VideoOverdue* and therefore we don't see a need to have a rule to particularly handle Extend relationships.

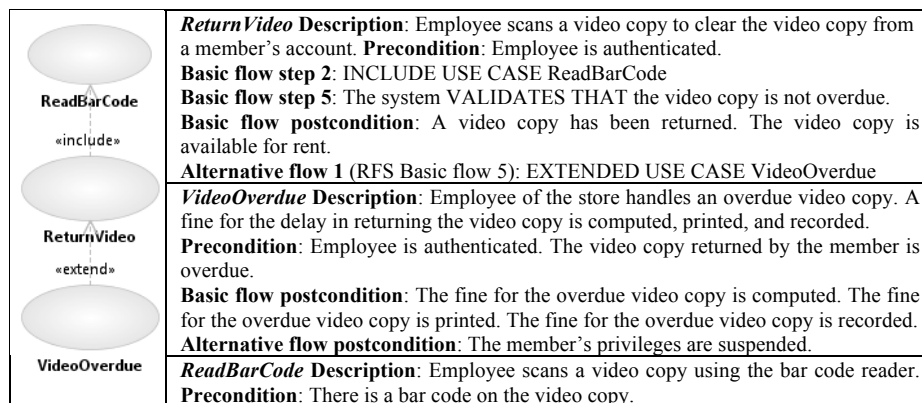


Figure 5 Example of the Include and Extend relationships among use cases

4.2 Transition to state machine diagrams for automated test generation

After the system-level state machine diagram is generated for a UCMOD, we need to follow the following steps to refine the state machine diagram so that it can be used as an input to automatically generate test cases.

1. We need to add missing transitions and states, remove extra states and transitions, or modify incorrect ones in the generated state machine, if required.
2. Second, we need to add state invariants using the Object Constraint Language (OCL) [19] for each state of the generated state machine based on the actual state variables of a system. For example, *NumberOfActiveCall* is a state

variable of Saturn that determines how many systems are currently in a video conference. This information is mandatory in state-based testing for automated oracle generation. However this information cannot be captured in the UCMod and is therefore missing in the state machine diagram generated by aToucan.

3. The third step is to map all the triggers in the state machine diagram to the actual API calls of the SUT so that the API of the system can be invoked while executing test cases generated from the state machine.
4. Last, it is also required to replace textual guard conditions of the generated state machine with corresponding OCL constraints, based on the state variables and/or input parameters of the triggers associated with the guards.

An effort estimate for these steps is provided on an industrial case study in Section 6.

5 Tool support

Our approach has been implemented as part of aToucan [7]. aToucan aims to automatically transform requirements given as a UCMod in RUCM into a UML analysis model including a class diagram, a set of sequence and activity diagrams, and a state machine diagram. It relies on a number of existing open source technologies. aToucan is built as an Eclipse plug-in, using the Eclipse development platform. UCMeta is implemented as an Ecore model, using Eclipse EMF [20], which generates code as Eclipse plug-ins. The Stanford Parser [21] is used as a NL parser in aToucan. It is written in Java and generates a syntactic parse tree for a sentence and the sentence's grammatical dependencies (e.g., *subject*, *direct object*). The generation of the UML analysis model relies on Kermeta [18]. It is a metamodeling language, also built on top of the Eclipse platform and EMF. The target UML analysis model is instantiated using the Eclipse UML2 project, which is an EMF-Based implementation of the UML 2 standard.

The architecture of aToucan is easy to extend and can accommodate certain types of changes. Transformation rules for generating different types of diagrams are structured into different packages to facilitate their modifications and extensions. Thanks to the generation of an Eclipse UML2 analysis model, generated UML models can be imported and visualized by many open source and commercial tools. Similarly, though UCSs are currently provided as text files, a specific package to import UCSs will allow integration with open source and commercial requirement management tools. More details on the design of aToucan can be found in [7].

We adapted the traceability model proposed in the traceability component (fr.irisa.triskell.traceability.model) of Kermeta [18] to establish traceability links. Details of the traceability model is discussed in [7].

6 Evaluation, discussion, and future work

Our goal here is to assess 1) whether the tool does generate system-level state machine diagrams based on UCMods, 2) whether our transformation rules are semantically complete, 3) whether our transformation rules lead to state machine diagrams that are syntactically correct, and 4) whether the automatically generated state machine diagrams can be refined by engineers to support MBT with reasonable effort. Regarding point 3, syntactic correctness means that a generated state machine

diagram conforms to the UML 2.2 state machine diagram notation. Regarding point 2, semantic correctness means that a generated state machine diagram correctly represents its UCMOD; all the constructs that are related to the transformation in the UCMOD are correctly transformed by following the transformation rules and no redundant model elements are generated.

Regarding the first three evaluation points, two software system descriptions (ATM and Elevator, called Banking System and Elevator System in [22]) were used to evaluate them. UCSs of these systems were re-written by applying RUCM so that they could be used as input of the transformations. These UCSs have also been used to generate class and sequence diagrams [7], and activity diagrams [23]. As the state machines provided in the textbook do not fully correspond to the UCMODs also provided in the same textbook, we cannot compare the state machine diagrams automatically generated by our tool with the ones provided in the textbook. However, we carefully examined our state machines, and we could verify that the generated state machines were syntactically correct and mostly but not entirely semantically complete. More importantly, this allowed us to identify the following limitations:

- The quality of UCSs has direct impact on the quality of the generated state machine diagrams. For example, correct preconditions and postconditions of UCSs are required in order to generate meaningful state machine diagrams. In our future work, we plan to propose guidelines on how to write UCSs using RUCM for the purpose of generating state machine diagrams, so that higher quality of UCSs will be defined and lead to higher quality state machines.
- In order to reduce redundant states and transitions, using advanced lexical analysis techniques such as WordNet [24] (an electronic lexical database) are needed to identify similar words and sentences, so as to eliminate redundancy among model elements in generated state machine diagrams. This is also one of our future research directions.
- Use case diagrams cannot model possible sequences of use case executions. However such information is sometimes needed in order to generate more complete and precise system-level state machine diagrams. For example, it is always an issue to identify the set of states where a state machine should transition from the start state (Section 4.1, rule 3). As shown in the state machine diagram in Appendix B, the automatically generated state machine diagram does not have any transition going from the start state to the other states of the state machine. The reason is that in order to create such transitions, we need to identify the possible sequences of executing use cases. Using our case study as example, we need to answer questions such as: is use case *MakeConferenceCall* executed before use case *StartPresentation*? A use case diagram cannot capture such information; therefore, using for example activity diagrams to capture such sequential constraints is needed. From a testing perspective, such information is required to generate feasible test cases from state machine diagrams. For example, a test case generated from a state machine diagram should correspond to an infeasible path, e.g., executing use case *StartPresentation* before use case *MakeConferenceCall*.

We also conducted an industrial case study to automatically generate a state machine diagram from the UCMOD of a video conferencing systems (Saturn) developed at Cisco Norway. The UCMOD contains seven use cases as shown in Figure

2 and the generated state machines contains five states and ten transitions. The automatically generated state machine was manually refined by one domain expert and one modeling expert working together by following the steps described in Section 4.2. More specifically, one transition from the `Start` state to state `The system is idle`, with trigger `PowerOn`, was manually added to the automatically generated state machine diagram (Appendix B). One transition from state `The system is idle` to state `The presentation is started`, with trigger `Start presentation`, was also added. This transition means that Saturn can start a presentation without being in any conference call. Notice that these two transitions were not automatically generated, because the required information was not described in the UCMOD. In total, it took around 40 minutes for them to complete the refinement process. In addition, they did not meet any difficulty in applying RUCM and they took less than three hours to complete the UCMOD of Saturn. aToucan took less than 200 seconds to generate the state machine diagram for Saturn. In the future, we plan to conduct controlled experiments to compare the quality of aToucan-generated state machines with the ones manually derived by engineers.

7 Conclusion

Over the last decade, model-based testing (MBT) was shown to be effective both in industry and academia. However, the success of MBT relies on developing complete and precise models. Developing such models from scratch can be a challenging task, especially when testers are not acquainted with modeling. To assist the initial modeling required for testing, we propose an approach to transform use case specifications into UML state machines, the most common notation used for MBT.

A use case modeling approach (RUCM) was proposed in [6] and was used in this paper to automatically generate UML state machines diagrams from textual use case specifications. This work is part of the aToucan approach [7], which aims to transform a use case model produced with RUCM into an initial UML analysis model that includes class and sequence diagrams [7], and activity diagrams [23]. We first evaluated our state machine transformations on two case studies from Gomaa's textbook [22]. We manually assessed the quality of generated state machines and found them largely consistent with the source use cases. In addition, we evaluated our approach on an industrial application, where we modeled use case specifications of a video conferencing system developed by Cisco Norway. These use case specifications were automatically transformed into initial state machine diagrams using our tool, which were then refined by a domain expert and a modeling expert to support test case generation. Our industry partner benefited not only from the executable test cases, but also from the system specification expressed as UML state machine diagrams and precise requirements expressed with RUCM, with all this for an overall cost of less than four hours, including documenting the use case model and refining the generated state machine diagram.

In the future, we are planning to provide detailed guidelines to help write use case specifications using RUCM. We are also planning to extend RUCM to allow the specification of use case execution sequences, which can further help in improving completeness of generated state machine diagrams. Empirical studies will be conducted to evaluate the quality of automatically generated state machine diagrams

compared to the ones manually generated by engineers.

8 Acknowledgement

This work was supported by a grant from Det Norske Veritas (DNV) and Simula Research Laboratory, Norway, in the context of the ModelME! Project. We are also grateful to Cisco Norway for their support and help on performing the industrial case study.

9 References

1. Shafique, M., Labiche, Y.: A Systematic Review of Model Based Testing Tool Support. Carleton University, Technical Report SCE-10-04
2. Neto, A.C.D., Subramanyan, R., Vieira, M., Travassos, G.H.: A survey on model-based testing approaches: a systematic review. WEASELTech. ACM, Atlanta, Georgia (2007)
3. Tandberg: Tandberg video conference system.
4. IEEE Std. 830-1998, IEEE Standard for Software Requirement Specification (1998)
5. Olsen, G.K., Oldevik, J.: Scenarios of traceability in model to text transformations. ECMDA-FA, Vol. 4530/2007. Springer Berlin / Heidelberg, Haifa, Israel (2007) 144-156
6. Yue, T., Briand, L.C., Labiche, Y.: A Use Case Modeling Approach to Facilitate the Transition Towards Analysis Models: Concepts and Empirical Evaluation. MODELS2009, Vol. 5795/2009. Springer Berlin / Heidelberg (2009) 484-498
7. Yue, T., Briand, L.C., Labiche, Y.: Automatically Deriving a UML Analysis Model from a Use Case Model. Simula Research Laboratory, Technical Report 2010-15 (2010)
8. Ali, S., Hemmati, H., Holt, N.E., Arisholm, E., Briand, L.: Model Transformations as a Strategy to Automate Model-Based Testing - A Tool and Industrial Case Studies. (2010)
9. Yue, T., Briand, L.C., Labiche, Y.: A systematic review of transformation approaches between user requirements and analysis models. *Requir. Eng.* (2010)
10. Insfrán, E., Pastor, O., Wieringa, R.: Requirements Engineering-Based Conceptual Modelling. *Requirements Engineering*, Vol. 7 (2002) 61-72
11. Some, S.S.: An approach for the synthesis of state transition graphs from use cases. Vol. Vol.1. CSREA Press, Las Vegas, NV, USA (2003) 456-462
12. Glinz, M.: Improving the quality of requirements with scenarios. The second world congress on software quality (2000) 55-60
13. Whittle, J., Schumann, J.: Generating statechart designs from scenarios. Proceedings of the 22nd international conference on Software engineering. ACM, Limerick, Ireland (2000)
14. Yue, T., Briand, L., Labiche, Y.: Facilitating the Transition from Use Case Models to Analysis Models: Approach and Experiments, Simula Research Laboratory, Technical Report (2010-12). (2010)
15. Bittner, K., Spence, I.: Use Case Modeling. Addison-Wesley Boston (2002)
16. OMG: UML 2.2 Superstructure Specification (formal/2009-02-04).
17. Brown, E.K., Miller, J.E.: Syntax: a linguistic introduction to sentence structure. Routledge (1992)
18. Kermeta: Kermeta metaprogramming environment. Vol. 2010. Triskell team
19. OMG: OCL 2.0 Specification.
20. Eclipse Foundation: Eclipse Modeling Framework. Vol. 2010
21. The Stanford Natural Language Processing Group. The Stanford Parser version 1.6
22. Gomaa, H.: Designing Concurrent, Distributed, and Real-Time Applications with UML. Addison-Wesley (2000)

23. Yue, T., Briand, L.C., Labiche, Y.: An Automated Approach to Transform Use Cases into Activity Diagrams. 6th ECMFA, Vol. LNCS 6138. Springer Heidelberg, Paris, France (2010) 337-353

24. Princeton University, WordNet: A lexical database for English, <http://wordnet.princeton.edu/>.

Appendix A Use case specifications of Saturn

The following use case specifications only contain the fields with important information to understand our approach due to space limitation.

Table 3 Use case Dial

<i>Brief Description</i>	User dials the system.
<i>Precondition</i>	The system is powered on.
<i>Basic flow steps</i>	1) User dials the system. Postcondition: The system is in a conference call.

Table 4 Use case MakeConferenceCall

<i>Brief Description</i>	The system is idle.
<i>Precondition</i>	User dials the system to make a conference call.
<i>Basic flow steps</i>	1) User dials the system. 2) The system makes a conference call to Endpoint. Postcondition: The system is in a conference call.

Table 5 Use case JoinConferenceCall

<i>Brief Description</i>	User dials the system to join a conference call.
<i>Precondition</i>	The system is in a conference call.
<i>Basic flow steps</i>	1) User dials the system. 2) The system VALIDATES THAT the maximum number of Endpoint to the conference call is not reached. 3) The system adds Endpoint to the conference call. Postcondition: The system is in a conference call.
<i>Specific Alt. Flow (RFS Basic flow 2)</i>	1) The system sends a message to User. 2) ABORT. Postcondition: The system is in a conference call.

Table 6 Use case StartPresentation

<i>Brief Description</i>	User wants to start the presentation.
<i>Precondition</i>	The system is in a conference call.
<i>Basic flow steps</i>	1) User requests the system to start a presentation. 2) The system sends the presentation to Endpoint. Postcondition: The presentation is started.

Table 7 Use case StopPresentation

<i>Brief Description</i>	User wants to stop the presentation.
<i>Precondition</i>	The presentation is started.
<i>Basic flow steps</i>	1) User requests the system to stop a presentation. 2) The system stops the presentation to Endpoint. Postcondition: The system is in a conference call.

Table 8 Use case DisconnectAll

<i>Brief Description</i>	User disconnects all Endpoint participating in a conference call.
<i>Precondition</i>	The system is in a conference call.
<i>Basic flow steps</i>	1) The system disconnects all connected EndPoints. 2) ABORT Postcondition: The system is idle.

Appendix B Generated state machine diagram for Saturn

