Combining Sequences and State Machines to Build Complex Test Cases

Dehla Sokenou[1] and Stephan Weißleder[2]

[1] GEBIT Solutions GmbH, Koenigsallee 75 b, 14193 Berlin
dehla.sokenou@gebit.de
[2] Humboldt-Universität zu Berlin, Rudower Chaussee 25, 12489 Berlin
weissled@informatik.hu-berlin.de

Abstract. Model-based testing is an important testing technology. The UML is a very popular modeling language. In this paper, we present a technique to utilize UML state machines in order to concatenate UML sequence diagrams and sketch corresponding coverage criteria. We show the relevance of our work by presenting an industrial case study in which sequence diagrams were combined with a state machine.

Key words: Unified Modeling Language, State Machines, Sequence Diagrams, Test Model Combination

1 Motivation

Functional testing is an important testing technique that is based on comparing the system under test (SUT) to specifications. Model-based testing is often used for functional testing. The test models are used as specifications. They provide information about input stimuli as well as the necessary oracle to deduce if a test case detects a fault.

Functional testing compares SUT and test models. Thus, SUT and test models must not be automatically derived from another and the creation of test models is often a manual task. In model-driven engineering, a requirement use case is often supplemented with a sequence diagram. Each sequence diagram represents a few possible behaviors of the SUT. Thus, there are only a few test cases for each use case. This supports traceability from requirements to test cases. For such reasons, sequence diagrams are very popular to model test cases. One issue about sequence diagrams, however, is that they just consist of a sequence of interactions without any notion of state. Thus, it is impossible to concatenate sequence diagrams, i.e. execute the described test cases consecutively.

State machines are a more complex means to model behavior than sequence diagrams. In contrast to a sequence diagram, a state machine is used to describe a large set of behavior traces: From most non-trivial state machines, a possibly infinite set of test cases can be derived. In such cases, coverage criteria are used as a stop criterion for test generation. Since the described behavior can be quite complex and the generated test cases are also determined by the used coverage criteria, the application of state machines makes traceability hard. However, state machines are based on state information, which can be utilized, e.g. to derive test oracle information based on state invariants.

In this paper, we propose the combination of state machines and sequence diagrams. The contribution is a technique to concatenate sequence diagrams by retracing their behavior in state machines and concatenating the corresponding transition sequences. For that, all possibly traversable transition sequences of the state machine matching to the information contained in the sequence diagrams have to be identified. The advantages are, e.g., the creation of longer sequences based on existing ones, the detection of faults that are undetected by the tests derived from simple sequence diagrams, or the satisfaction of stronger coverage criteria on the state machine. Additionally, this approach provides test oracles in the form of state constraints to test cases derived from sequence diagrams.

We use an automated teller machine (ATM) from an industrial case study to clarify our approach. In this case study, sequence diagrams are used to describe the behavior of the ATM for one customer including, e.g. checks for the correct PIN. The sequence diagrams were used to derive a state machine that contains the behavior of all sequences. Furthermore, the state machine was extended manually and contains additional loops that, e.g. allow the repeated usage of the ATM for several customers. While the sequence diagrams describe the interaction of several objects, the state machine only describes the behavior of the ATM. Behavior of other systems like AR will not be considered in this paper.

This paper is structured as follows. The next section contains the related work. Section 3 comprises the proposed combination of state machine and sequence diagrams. The case study is presented in Section 4. The final section concludes and provides future prospects.

2 Related Work

There has been a lot of work about model-based testing much of which is condensed in [11]. Especially, state machines and sequence diagrams of the Unified Modeling Language (UML) [4] are often used to model test cases. As one example, Nebut et al. [3] derive test cases from contracts such as use cases and sequence diagrams. As another example, Abdurazik and Offutt provide an approach for automatic test generation from state machines [5]. In contrast to that, we aim at the combined use of both diagrams to generate test suites.

There has also been work about the combination of sequence diagrams and state machines. From the very beginning, it was clear that sequence diagrams can describe single transition sequences of a state machine. Bertolino et al. [1] combine both diagrams to derive "reasonably" complete test models to achieve early results for partially modeled systems. Sokenou [9] and Nagy [2] showed furthermore, that the state machine's start states to execute sequence diagrams can be very important. We extend these approaches by identifying transition sequences instead of start states that are matching to sequence diagrams. Additionally, we combine several sequence diagrams by matching start and end sequences of the corresponding transition sequences to build new and more complex sequences, and we propose coverage criteria based on sequence diagrams.

3 Combination of State Machine and Sequence Diagram

Sequence diagrams are often used to describe test cases manually (see e.g. UML Testing Profile and TTCN-3 [7]) where a message to a given lifeline is considered as test input to the corresponding object under test. Due to missing state information, however, the sequences cannot be concatenated. State machines describe a possibly infinite set of test cases but the proper selection of concrete test cases is a complex task. In this section, we describe how to concatenate sequence diagrams by retracing them as transition sequences in state machines and how to take advantage of this concatenation. For that, we denote several states as follows: The *initial state* is the initial state of the state machine as defined in the UML specification [4, page 521]. Other states are used to refer to the start or the end of an interaction sequence described in a sequence diagram: A *start state* of a sequence is a state in the state machine that allows to start the execution of the sequence from. The corresponding target state of this execution is called *end state.*

3.1 Advantages of Concatenating Sequence Diagrams

Before describing the concatenation of sequence diagrams, we motivate this approach by listing some of the resulting advantages. First, the set of possible start states of a sequence seq_1 is in many cases disjunct to the set of the state machine's initial states. In order to execute seq_1, we have to find a sequence seq_2 from one of the state machine's initial states to one of seq_1's start states s. We call such a seq_2 an "initialization sequence“ for seq_1 if seq_2's set of start states contains at least one of the state machine's initial states and s is the end state of one of seq_2's transitions. Since there are often several sequences whose possible start states overlap with the state machine's initial states, all these sequences can be used as initialization sequences for other sequences. So, the first advantage of concatenating sequence diagrams is to reuse existing sequence diagrams as initialization sequences for sequence diagrams that cannot be executed from an initial state.

Second, the concatenation of sequence diagrams results in longer and possibly fewer test cases. In some environments like embedded systems, the initialization of test cases results in higher costs than the test execution. Thus, the combination of many test cases into a few long ones can save expenses.

Third, as we will show in our case study, the concatenation of sequence diagrams can result in the detection of faults that are undetected by the execution of single sequence diagrams. Concatenated sequence diagrams can be used to test the concatenated behavior of several sequences as well as their repeatability.

Finally, the presented possibility of sequence diagram concatenation introduces new means of quality measurement for executing sequence diagrams. Until now, the sole execution of a sequence diagram can be measured, e.g. with the coverage criterion *All-paths sequence diagram coverage* [11, page 122]. Furthermore, the identification of matching (initialization) transition sequences allows the sequential execution of sequence diagrams. We would call the corresponding coverage criterion *All-Context-Sequences.* For the concatenation of sequence diagrams, we can introduce further coverage criteria that are similar to existing transition-based coverage criteria: For instance, *All-Sequence-Pairs* (the concatenation of all pairs of sequence diagrams), *All-n-Sequences* (the concatenation of all n-tuples of sequence diagrams), or *All-Sequence-Paths* (all sequence diagram concatenations of any length). These new coverage criteria might be useful for evaluating the test suites derived from a state machine and a set of sequence diagrams. Their use, however, has to be evaluated in case studies.

3.2 Formal Definitions

There are many different kinds of state machine definitions (cf. [8, 10]). In this paper, we stick to the definitions of state machines as presented in the UML 2.1 specification [4]. We sketch basic elements of this definition. A state machine $sm \in SM$ is a set of regions Reg and pseudo states PS. Each region contains a set of vertices $Vert$ (i.e. states and pseudo states) and a set of transitions $Trans$. Transitions $trns \in Trans$ connect vertices – they reference a corresponding source and target vertex ($trns.source$, $trns.target$). Each transition contains instances of trigger T_{sm}, guard G, and effect E. The set of all transitions in state machine $Trans_{sm}$ is derived as union of transition sets from all regions. Based on this, we denote $TransSeq$ as the set of all transition sequences of a state machine. Each transition $trns \in Trans$ has a label $t[g]/[e]$ where $t \in T_{sm}$ is the trigger, $g \in G$ is the guard $trns.guard$ of $trns$, and $e \in E$ is the effect $trns.effect$ of $trns$. Both elements g and e are of type OCL. We only consider deterministic state machines to have an unambiguous mapping from triggers to transitions.

A sequence diagram is defined as a 5-tupel $SD = (L, M, O, \Lambda_{sd}, T_{sd})$. The set L represents the life lines of all objects in the given sequence. The relation $M = L \times \Lambda_{sd} \times L$ includes all messages sent between life lines. Each message $m \in M$ has a label $t(para) \in \Lambda_{sd}$ where $t \in T_{sd}$ is the trigger of the message and $para$ represents all parameters of t. $O = M \times M$ defines the ordering of messages. A message m_1 is called before a message m_2 in a sequence iff $(m_1, m_2) \in O$. In our scenario, O is total, i.e. there are no asynchronous messages.

Triggers are method calls or events send to an object. A sequence diagram's set of triggers $T_l \subset T_{sd}$ for incoming messages of a lifeline l is a subset of all triggers T_{sm} of the corresponding state machine $sm \in SM$ if sm is associated to l, i.e. describes the behavior of the object of l. A message $m \in M$ with the label $t(para)$ can be executed in a given state $s \in Vert$ iff $\exists trans \in Trans : trans.source = s$ and $trans.guard$ is satisfied by the current attribute and parameter value assignment resulting from the already executed messages. Our definition conforms to UML protocol state machines and considers only explicitly modeled transitions.

3.3 Concatenate Sequence Diagrams

This section contains the descriptions of how to concatenate sequence diagrams by concatenating corresponding transition sequences of a state machine. For that, we have to derive state machine transition sequences from sequence diagrams. We introduce the function $\mathit{findTransitionSequences} : SD \times SM \rightarrow \mathscr{P}(TS)$ that produces transition sequences for each combination of sequence diagram and state machine. Afterwards, we show how to concatenate the transition sequences.

In [9], we described a method to derive a set of possible start states for the execution of behavior defined in sequence diagrams. In contrast, this paper is focused on deriving and comparing transition sequences instead of single states. Thus, we present an algorithm to derive transition sequences from state machines that reflect the described behavior of sequence diagrams. Fig. 1 shows the algorithm of the corresponding function *findTransitionSequences*. Due to reasons of conciseness, the pseudocode leaves out some aspects of transition matching such as transition guards, post conditions or state invariants. Nevertheless, we are aware that tracing the current system state (i.e. system attribute value assignment) is important to determine important aspects of transition matching, e.g. the satisfaction of transition guards.

```
findTransitionSequences (sequenceDiagram, stateMachine) {
  sequences = empty set; // return value
  msg = first message of sequenceDiagram;
  startStates = all states of stateMachine with outgoing transitions
                triggered by msg;
  for each(s in startStates) {
    tmpS = s;
    transitionSequence = empty sequence;
    for(i = 0; i < number of messages of sequenceDiagram; ++i) {
      msg = sequenceDiagram.messages[i];
      if(tmpS has outgoing transition t triggered by msg) {
        tmpS = target state of t;
        add t to transitionSequence;
      }
      else {
        transitionSequence = empty sequence;
        break;
      }
    }
    if(transitionSequence is not empty) {
      add transitionSequence to sequences;
    }
  }
  return sequences;
}
```

Fig. 1. Algorithm for detecting all transition sequences for a sequence diagram.

For each sequence diagram, we retraced its described behavior as a sequence of transitions in the state machine. In this section, we use these transition sequences to combine several sequence diagrams: For two transition sequences $ts_1, ts_2 \in TS$ with ts_1 is assumed to be executed before ts_2, we consider three cases: 1) ts_1 does not include a transition whose target state is the start state of ts_2. In this case, both transitions cannot be concatenated. 2) The target state of the last transition in ts_1 is equal to the source state of the first transition in ts_2 and 3) an end sequence ts_{1B} of ts_1 is equal to a start sequence ts_{2A} of ts_2 (see Fig. 2). For the second and the third case, ts_2 can be executed after ts_1 (without executing the overlapping transitions in ts_{1B}/ts_{2A} twice).

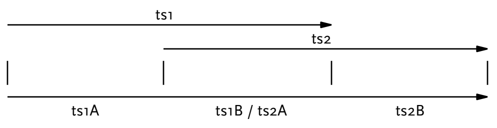

Fig. 2. Overlapping transition sequences.

4 Case Study

In this section, we present a scenario in which sequence diagrams were derived from requirements of an automated teller machine (ATM) and applied to create test cases for the ATM. Furthermore, these sequence diagrams are composed to a state machine. Originally, test cases were only derived directly from requirements, not from models. In the following, we show how to use our approach to generate more complex test cases for such scenarios.

The state machine is shown in Fig. 3. Transitions of state machine are numbered so it is easier to describe transition sequences in the following. A possible behavior of a customer that enters his PIN (4 digits) and withdraws money is described for instance by the following transition sequence: (4, 14, 15, 15, 15, 15, 24, 23, 20, 19, 13, 8, 2) (cf. Fig. 3). In the figure, AC stands for Account Check and AR stands for Authorization System. Note that the state machine describes just the behavior of the ATM. Thus, some messages of the sequence diagrams are not included in the state machine.

All sequence diagrams are directly derived from requirements use cases. Example sequences are shown in Fig. 4, 5 and 6. Sequence 1 is an initializing sequence as it can be executed in the initial state of the state machine, state *Idle*. The message *ec_inserted* can only be executed in state *Idle*, thus the algorithm starts with transition *4:ec_inserted* in the state machine. Following the algorithm in Fig. 1, transition sequences for Sequence 1 are: $transseq_{s1.1}$ = (4, 14, 15, 15, 15, 15, 24, 23) and $transseq_{s1.2}$ = (4, 14, 15, 15, 15, 15, 24, 21). The last transitions of both sequences depend on the values of the attributes

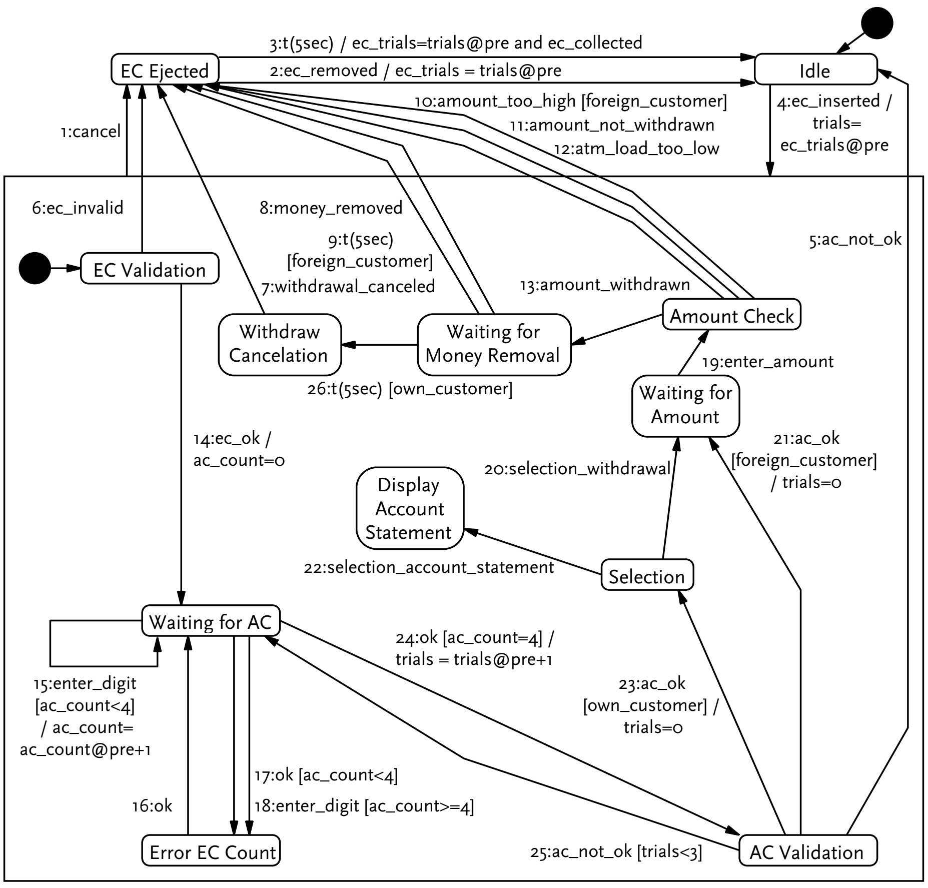
EC Ejected
Idle
3:t(5sec) / ec_trials=trials@pre and ec_collected
2:ec_removed / ec_trials = trials@pre
10:amount_too_high [foreign_customer]
11:amount_not_withdrawn
12:atm_load_too_low
1:cancel
4:ec_inserted / trials= ec_trials@pre
6:ec_invalid
8:money_removed
9:t(5sec) [foreign_customer]
7:withdrawal_canceled
5:ac_not_ok
EC Validation
13:amount_withdrawn
Amount Check
Withdraw Cancelation
Waiting for Money Removal
26:t(5sec) [own_customer]
19:enter_amount
Waiting for Amount
14:ec_ok / ac_count=0
21:ac_ok [foreign_customer] / trials=0
20:selection_withdrawal
Display Account Statement
22:selection_account_statement
Selection
Waiting for AC
24:ok [ac_count=4] / trials = trials@pre+1
15:enter_digit [ac_count<4] / ac_count= ac_count@pre+1
23:ac_ok [own_customer] / trials=0
17:ok [ac_count<4]
16:ok
18:enter_digit [ac_count>=4]
Error EC Count
25:ac_not_ok [trials<3]
AC Validation

Fig. 3. State Machine of ATM

own_customer and *foreign_customer*. For Sequence 2, only one transition sequence can be found: $transseq_{s2} = (20, 19, 13, 8)$. There is also only one transition sequence for Sequence 3: $transseq_{s3} = (8, 2)$.

The target state of the last transition of $transseq_{s1.1}$ is the same as the source state of first transition of $transseq_{s2}$. Thus, both are concatenated to build a new scenario and a resulting new transition sequence $transseq_{s1.1,s2} =$ (4, 14, 15, 15, 15, 15, 24, 23, 20, 19, 13, 8). As the transition sequence $transseq_{s2}$ ends with the same transition as $transseq_{s3}$, also Sequence 3 can be concatenated to the others. One of the resulting transition sequences is $transseq_{s1.1,s2,s3} =$ (4, 14, 15, 15, 15, 15, 24, 23, 20, 19, 13, 8, 2). Corresponding to the transition sequences, we concatenated the sequence diagrams and created longer test cases. Fig. 7 shows the resulting sequence diagram.

As we expected, several faults can be detected by longer sequence diagrams. For instance, the combined sequence is the only sequence in the case study that

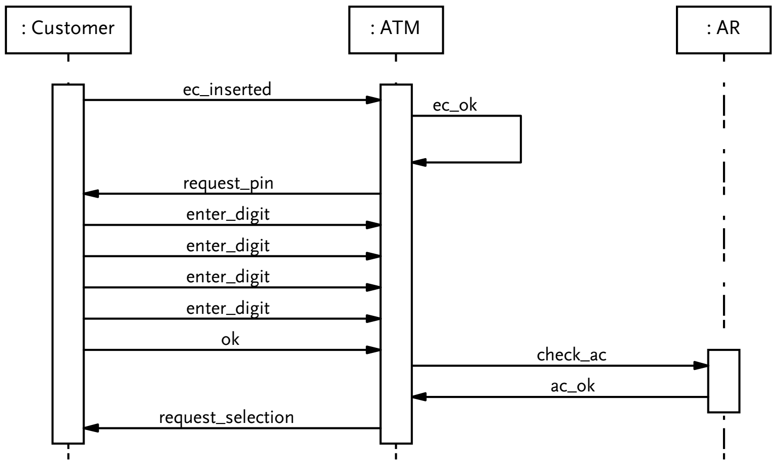

Fig. 4. Sequence Diagram 1

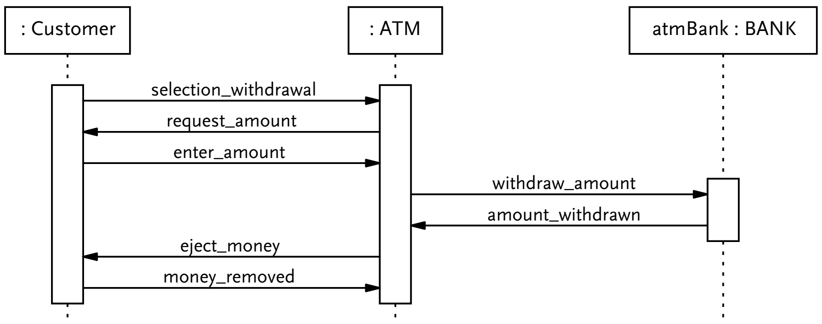

Fig. 5. Sequence Diagram 2

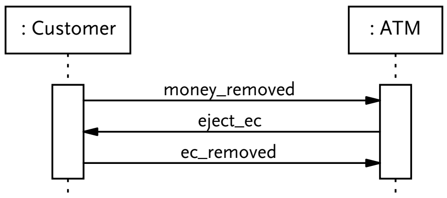

Fig. 6. Sequence Diagram 3

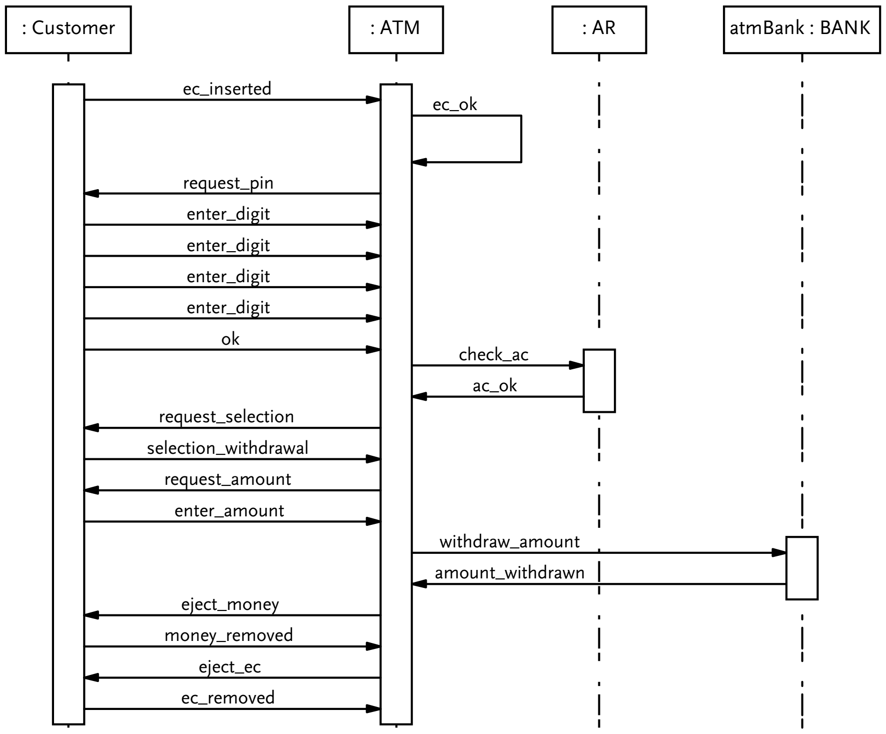

Fig. 7. Concatenated Sequence Diagram

contains a scenario from inserting EC card until money and EC removal. Furthermore, we found a sequence that was not covered by the original sequence diagrams but can be derived by combining sequence diagrams: The original sequence diagrams only describe that a customer enters a wrong PIN three times without removing the EC card in between. However, there is no sequence diagram for a customer that three times inserts an EC card, enters the PIN incorrectly just once, and cancels the operation afterwards. With the repetition of several complete scenarios for interactions of customer and ATM, such scenarios can be covered. We found these improvements by manual inspection. It would be interesting to automate this approach to identify and evaluate further advantages.

5 Conclusion and Outlook

In this paper, we presented an approach to combine sequence diagrams by retracing their described behavior as transition sequences in a state machine. We presented a concrete algorithm, listed several advantages of this approach, e.g. mentioned possible resulting coverage criteria, and showed the applicability of our approach for an industrial case study.

Sequence diagrams are used to describe typical scenarios. They often do not define conditions for the initial state of execution and describe only parts of a scenario. Thus, the presented concatenation of sequence diagrams is a good way to create more complex test cases while avoiding to initialize each sequence separately before its execution. This concatenation can be automated and, thus, no additional manual effort is necessary. Additionally, the proposed combination of two diagrams allows to define new coverage criteria based on both diagrams.

There are some points left to discuss. First, combining sequences can result in a lot of new test cases. More complex test cases can reduce test effort but increase the effort to find errors. To reduce complexity, the number of these test cases might be reduced using the proposed coverage criteria. Second, we have to define limitations on the algorithm. It seems to make no sense to combine sequence diagrams that overlap in all transitions except one or to generate only one combined test case from all sequences. What is the maximum overlap that should be taken in consideration? What is the minimum number of test cases?

In the future, we plan to implement the presented approach, e.g. as an extension of the tool ParTeG [6]. We want to use the tool to automatically create test cases for concatenated sequence diagrams. Here, we will take also the proposed new coverage criteria on sequence diagrams into account.

References

1. A. Bertolino, E. Marchetti, and H. Muccini. Introducing a Reasonably Complete and Coherent Approach for Model-based Testing. *Electr. Notes Theor. Comput. Sci.*, 116:85–97, 2005.
2. R. Nagy. Bedeutung von Ausgangszuständen beim Testen von objektorientierter Software. In *CoMaTech '04*, Trnava, Slowakei, 2004.
3. C. Nebut, F. Fleurey, Y. L. Traon, and J.-M. Jézéquel. Requirements by Contracts allow Automated System Testing. In *ISSRE'03*, pages 17–21, Denver, CO, USA, 2003.
4. Object Management Group. Unified Modeling Language (UML), version 2.1, 2007.
5. J. Offutt and A. Abdurazik. Generating tests from UML specifications. In R. France and B. Rumpe, editors, *UML'99 - The Unified Modeling Language. Beyond the Standard.*, volume 1723 of *LNCS*, Fort Collins, CO, USA, 1999. Springer.
6. S. Weißleder. ParTeG (Partition Test Generator). http://parteg.sourceforge.net.
7. I. Schieferdecker and J. Grabowski. The Graphical Format of TTCN-3 in the context of MSC and UML. In *Telecommunications and beyond: The BroaderApplicability of SDL and MSC (SAM'2002)*, volume 2599 of *LNCS*, Rosslyn, VA, USA, 2003. Springer.
8. D. Seifert, S. Helke, and T. Santen. Test Case Generation for UML Statecharts. In *Perspectives of System Informatics*, volume 2890 of *LNCS*, Novosibirsk, Russland, 2003. Springer.
9. D. Sokenou. Generating Test Sequences from UML Sequence Diagrams and State Diagrams. In *MOTES 2006*, Dresden, Germany, 2006.
10. D. Sokenou. *UML-basierter Klassen- und Integrationstest objektorientierter Programme (german).* PhD thesis, Technische Universität Berlin, Germany, 2006.
11. M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2006.