Towards an Automatic Test Case Decomposition by Means of System Decomposition

Marcel Ibe and Andreas Rausch

Clausthal University of Technology
Email: marcel.ibe@tu-clausthal.de, andreas.rausch@tu-clausthal.de

***Abstract*—Quality assurance takes a huge effort during a software development project. Especially the generation of test cases and test data but also the execution, analysing the results and the maintenance consumes a lot of resources. Model-based testing tries to reduce the effort by automating several testing activities. In this paper, an approach for test case decomposition by means of system decomposition is introduced. The system's structure is described by composition structure diagrams, the systems behaviour by state charts. By transferring structural decomposition steps and use of the additional behavioural descriptions, existing test cases can be adapted to the refined system description automatically.**

Keywords–Model-Based-Testing, Test-Case-Decomposition, Sequence Diagram

I. INTRODUCTION

Testing is one of the most widely used practices to ensure high quality of software systems. At random the real behaviour of a system or a component is compared with the desired one by in advanced defined test cases. It takes up a very big share of the effort of a software development project [1]. Model-Based Testing (MBT) uses models to support testing activities, for example, by generating test cases automatically. By means of a test case specification a finite set of test cases (a test suite) is selected, that will be executed on the system [2]. According to the test case specification, the resulting test suite can contain a very large number of test cases. Since the number of test cases in a test suite is one of the factors that have a significant influence on total testing costs [3], one would like this number to be as small as possible. However, an afterward reduction of the test suites size can be both a hard problem [4] [5], as well as have an adverse effect on the quality of the test suite [6]. Several studies suggest that manually derived test cases provide an alternative to automatically generated ones. For example, Pretschner et al. [7] have found that not the size of a test suite but rather the basis of the test case generation reveals about the fault-finding ability. They observed that test suites containing a much higher number of automatically generated test cases detect only a few more errors than fewer, hand-crafted test cases. Marques et al. [8] compared manual ad hoc tests with automatically generated ones. They confirmed the observation, that manually derived test suites are usually smaller but not less effective in finding bugs. The study even gave evidence that manually derived test suites could find more major bugs. This suggests that a small test suite containing manually derived test cases provides an alternative solution to automatic test case generation. Although, the manual test case derivation has the main disadvantage of higher effort, it is nevertheless feasible in limited range of the software development process like user acceptance testing. If the strong fault-finding ability with the small test suite size could be transferred to other testing levels, the total costs of testing could be reduced.

The contribution of this paper is to introduce an approach that allows reusing manually derived test cases at different testing levels by automatically decompose these test cases analogously to the decomposition of the system under test (SUT). In this way, test suites for component or unit testing can be created that also have a strong fault-detection capability but contain only a small number of test cases.

In this paper, we present an approach for decomposing test cases by means of system decomposition to create new test cases for testing the SUT at different levels of decomposition. The next section introduces the overall approach, a running example and specifies the requirements to test case decomposition at the example. In Section 3, the test case decomposition is described in detail. Section 4 gives an overview over related work. In the last section, the results and plans for future work are presented.

II. OVERALL APPROACH

In this section, we introduce our approach for decomposing test cases. Figure 1 shows schematically how the test case decomposition can be applied during a software development project. Based on the requirements of the customer a first specification of the system is created manually. At the same time, test cases for user acceptance testing are derived. These test cases are black box tests that do not consider the internal structure but only the systems behaviour. During the development, the specification is getting more and more detailed by decomposing the system into components and describing the behaviour of these new components. Now, instead of creating new test cases for testing these components the already existing test cases can be used by enriching them with the new information about the internal structure and behaviour of the system. For every decomposition step at the systems specification these decompositions are transferred automatically to the test cases and so they are adapted and able to test the new defined system components. After starting implementing the specified components of the system the decomposed test cases can be used to test the components against the specification.

For applying our approach, a structural description of the SUT and its internal structure consisting of components and their ability to interact with each other is needed. We use an adapted version of UML Composition Diagrams to describe the internal structure of the systems and its components. For the behavioural description of the SUT and its internal

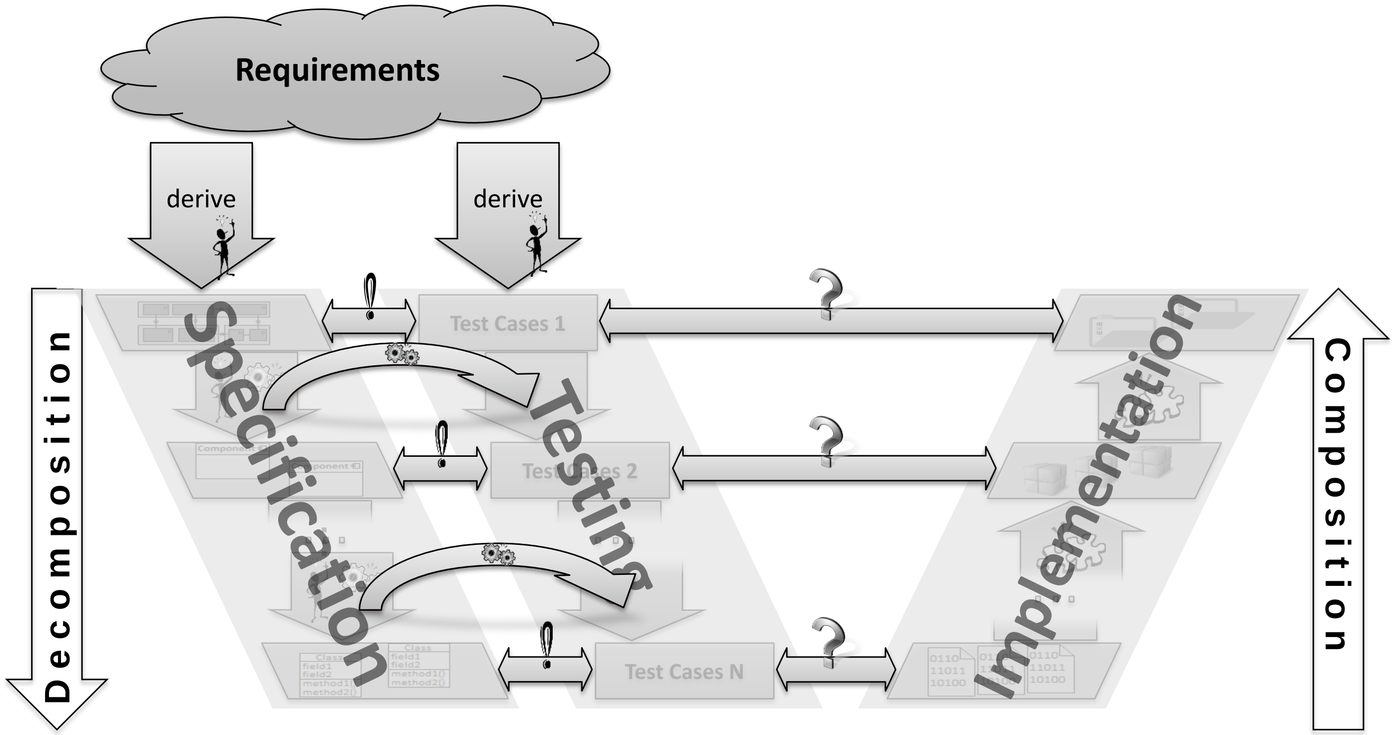

Figure 1. Schematic illustration of the test case decomposition approach applied during a software development process

components UML State Machines are used as graphical representation. At last we use an adapted version of UML Sequence Diagrams to describe the test cases.

In the next subsections, we introduce a running example and explain the diagrams used for describing the system's structure, behaviour and test cases in detail.

A. Running Example

To illustrate our approach, we use a small example of a central locking system (CLS) for cars which is inspired by the example used by Krüger et al. [9] and is shown in Figure 2. The upper image of Figure 2 shows that the CLS defines an interface with the four signals *lock*, *unlock*, *locked* and *unlocked* which are used for the communication of the CLS with its environment. The lower image of Figure 2 shows the state-based behaviour of the CLS. There are the initial state *unlocked* and the state *locked*. The CLS can switch from unlocked to locked state when receiving the signal *lock*. Additionally the signal *locked* is sent and vice versa when receiving the signal *unlock* and sending the signal *unlocked*.

B. Structural Description

The graphical representation of the structural description of the SUT and its internal components and subcomponents is based on UML Composition Structure Diagrams [10] (CSD). The structural decomposition of the CLS is shown in the left image of Figure 3. The CLS consists of the parts *control* and *motors*. The *control* part contains exactly one instance of the *control* component and the *motor* part contains two to five instances of the *motor* component. The parts can communicate with each other via the *MotorControl* interface, which is provided by the *motor* component and defines the

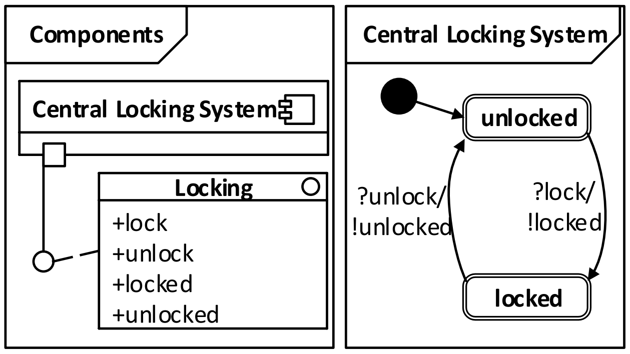

Figure 2. Structural (upper) and behavioural (lower) description of the central locking system

three signals *up*, *down* and *ready*. Furthermore, the *control* part can communicate with the environment of the CLS via the *Locking* interface, which is provided by the CLS itself and defines the signals *lock*, *unlock*, *locked* and *unlocked*. The *motor* and *control* components could be decomposed in the same way.

C. Behavioural Description

The behavioural description of the SUT and the components defined in the CSD is based on UML Statecharts [10] (SC). In addition to the components, the signals to be used in the SC are defined by the interfaces in the CSD.

The upper part of the right image of Figure 3 shows the behaviour of the CLS, which was already explained in Subsection II-A. In the lower part SCs, for *control* component

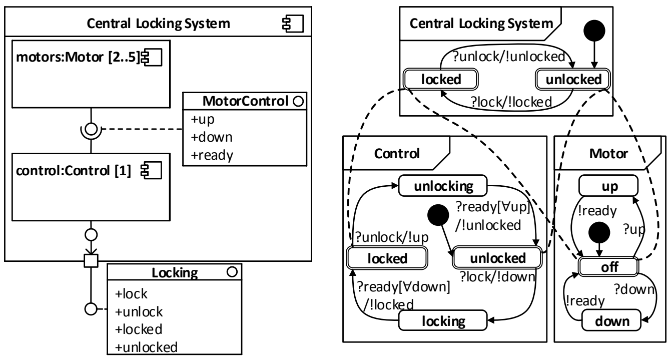

Figure 3. Structural (left) and behavioural (right) descriptions of the CLS and its components

and *motor* component are shown. For *control* component the four states *unlocked*, *locking*, *locked* and *unlocking* are defined. The double framed states *unlocked* and *locked* are so-called stable states. That means, the component may stay in this state for an unlimited period of time. *Unlocked* state is the initial state of this component. After receiving the signal *lock* being in this initial state it sends the *down* signal and switches to *locking* state. This state can only be left if every receiver of the *down* signal responses with a *ready* signal. Then this component switches to *locked* state after sending the *locked* signal. Switching the states to *unlocking* and *unlocked* states are performed analogous with the *unlock*, *up*, *ready* and *unlocked* signals. For *motor* component there are three states defined: The initial stable state *off* and *up* and *down* states. After receiving *up* respectively *down* signal, *motor* component switches to *up* respectively *down* state. Since, they are both no stable state *motor* component has to leave them and switch back to *off* state and consequently sends a *ready* signal.

The statecharts shown in Figure 3 describe the behaviour of components at two different decomposition levels of the SUT: The CLS at the top-most level and one level below *control* and *motor* component. For tracing the refinement between two levels a relation that assigns a set of states of the lower level to every stable state of the upper level has to be defined. If a SC of a lower level has both stable and not stable states the assigned set of states must contain at least one state from this SC. Else the relation to this lower level SC is optional. This relation will be used later to ensure that the components of a decomposed test case are in states that corresponds to a state of the component which was tested by the former test case. If there is no such relation for one or more SCs, the final states of the components in the decomposed test case do not have to be considered. In the CLS example *locked* state from *control* component and *off* state from *motor* component are assigned to *locked* state from CLS and *unlocked* state from *control* component and *off* from motor component are assigned to *unlocked* state from CLS.

D. Test Case Description

For describing the test cases, diagrams are used that are based on UML Sequence Diagrams [10] (SD). Every test case is described by one SD and consists of exactly one lifeline, which represents an instance of the component to be tested by this test case, and signals (messages) that are sent between the lifeline and its environment during this test case. This lifeline also contains the states that the corresponding component is in. The sent messages correspond to the signals that are sent or received during the transitions in the SCs and can cause a state change of the lifeline.

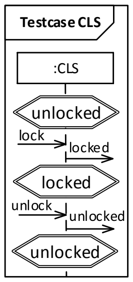

Figure 4. Graphical representation of a test case for CLS component

Figure 4 shows one test case for the CLS. Starting with the CLS lifeline in *unlocked* state receiving the *lock* message from its environment which causes sending the *locked* message and a switch to *locked* state. Then receiving the *unlock* message causes the CLS to send the *unlocked* message and a state change back to *unlocked* state. This behaviour corresponds to the SC for the CLS shown in Figure 2.

III. Test Case Decomposition

In this section, the test case decomposition is described in detail and its application is shown at the running example. The test case decomposition contains of two stages: Test case extension and test case partition. During the first stage, the test case is extended using the information of the systems decomposition from the CSD and the SCs of the new subcomponents. Hence, the test case is enriched with information about the internal communication between the subcomponents. Within the second stage the extended test case is partitioned into several test cases that respectively test one of the subcomponents of the component to test. These two stages of the test case decomposition are described in detail during the next subsections.

A. Test Cases Extension

In the first stage, a given initial test case is getting extended. That means that the structural decomposition and the new information about the behaviour of the decomposed components are transferred to the initial test case. The initial test case contains exactly one lifeline, which represents an instance of the component to be tested, the messages that are sent between the lifeline and its environment and the lifelines states during the test case. At first the new structural information is added to the initial test case by replacing the initial lifeline by lifelines for all instances that compose the initial component as describes in the CSD. After that the new behavioural information are added. This is done by retaining the initial messages from and to the environment and add the new messages which are sent between the new lifelines. Figure 5 shows the test case extension as pseudocode.

Using the example test case illustrated in Figure 4 we perform the test case extension. Figure 6 shows the test case at several intermediate steps during and after its extension. The algorithm gets as input the test case tc to be extended

```
   Data: test case $tc$ to be extended, lifeline $l_{tc}$ of component to
         be tested by $tc$, list of messages $M$ sent in $tc$
   Result: extended test case $tc$
1  Replace $l_{tc}$ by lifelines $L_{ext}$ for subcomponents
   List of free messages $M_f$ := $\emptyset$
   foreach $m \in M$ do
2      if $l_{tc}$ receives $m$ then
3          Mark $m$ as free incoming Message
4      else
5          Mark $m$ as free outgoing Message
6      $M = M \setminus m$
       $M_f = M_f + m$
7  while transition with ANY-Trigger available or $M_f \neq \emptyset$ do
8      while transition $t$ with ANY-Trigger available do
9          fire transition $t$
           $M_n$:= list of messages received due to fire $t$
           $M_f = M_n + M_f$
           update state of corresponding lifeline

10     if $M_f \neq \emptyset$ then
11         $m_i \in M_f$ first free incoming message
           $L$:=set of lifelines, that can receive $m$
           if $L \neq \emptyset$ then
12             foreach lifeline $l \in L$ do
13                 Create copy $m'$ of $m_i$
                   Bind $m'$ to $l$ and mark $m'$ as bound
                   $M_n$:= list of messages received due to $m'$
                   Update state of $l$
                   $M_f = M_n + M_f$
14         else
15             foreach free outgoing message $m_o \in M$ do
16                 if $m_i == m_o$ then
17                     Bind $m_o$ to sending lifeline $l$ of $m_i$
                       Update state of $l$
                       $M_f = M_f \setminus m_o$
18         Delete $m_i$ from $tc$ and $M_f$
```

Figure 5. Pseudocode for test case extension

containing the lifeline l_{tc} and a list of messages M that are sent between l_{tc} and its environment. After its execution, the algorithm returns the extended test case tc.

At the first step the algorithm replaces the lifeline l_{tc} by a set L_{ext} with lifelines in their initial state for every instance of a subcomponent as defined in the CSD of the component to test. For variable multiplicities of a part in the CSD, the lowest valid value greater zero is selected as number of instances. After this step the test case is adapted to the new structural information. Now, the empty list M_f is defined to collect all *free messages*. *Free messages* are messages, that do not have a sending (*free outgoing message*) or receiving (*free incoming message*) lifeline. In the following loop (lines 3 - 9) all messages in M are marked either as *free incoming* or *free outgoing message* depending on whether they were received or sent by lifeline l_{tc} and added to the end of M_f. The former order of the messages is retained. The current state of the test case is shown in the upper left image of Figure 6. The former lifeline *:CLS* was replaced by a *control* lifeline and two *motor* lifelines as two is the lower bound of part motors (see Figure 3). The *lock* and *unlock* messages were marked as *free incoming message* so one of the available lifelines can receive them. The *locked* and *unlocked* messages were marked as *free outgoing messages* and a lifeline has to be found that sends these messages.

The next loop (lines 10 - 31) is executed while there are transitions left that can be fired, more precisely transitions without a trigger (ANY-trigger) are available from the current states of the lifelines in L_{ext} or there are free messages left. First, all transitions with ANY-Trigger are fired (lines 11 - 15). Thereafter, new messages that are received due to firing these transitions are added at the top of M_f and the states of the corresponding lifelines are updated. If there is no transition with ANY-trigger left the first free incoming message $m_i \in M_f$ is bound to suitable lifelines in L_{ext}. Thereto, the corresponding statecharts of the lifelines in L_{ext} are searched for transitions that have the current state of the lifeline as source and m_i as trigger. For every possible receiver lifeline $l \in L_{ext}$ a copy m' of m_i is created and bound to l, i.e., l now receives the copy m' and m' is marked as bound (no longer a free message). Due to firing a transition, new free incoming messages M_n can be sent and are added at top of the list M_f after updating the state of lifeline l. If there are no possible receiving lifelines all remaining free outgoing messages m_o are compared to m_i and if there is a m_o with the same signal as m_i, m_o is bound to the lifeline sending m_i and m_o is removed from M_f. At the end the message m_i is deleted from the test case tc and the list M_f. In our example, there are no possible transitions with ANY-trigger, but there are free incoming messages. The first one that is chosen is the *lock* message. A copy of this message can be bound to the *control* lifeline because it is in *unlocked* state and there is a transition from *unlocked* state with trigger *lock* in the corresponding statechart. As a result of firing this transition, the new free incoming message *down* is added at top of M_f and *control* lifeline switches to *locking* state. Since, there are no more possible receiver lifelines the *lock* message can be deleted from the testcase. The upper central image of Figure 6 illustrates the current state of the test case.

The new *down* message can now be bound at the two *motor* lifelines. So there are two copies of this message, which are bound to these lifelines, causing them to switch to *down* state and the original message can be deleted. The current state of the test case is shown in upper right image of Figure 6. Now, there are two ANY-trigger transitions available which send the new free incoming *ready* messages and switch the *motor* lifelines back to *off* states. After binding these two *ready* messages to *control* lifeline it receives a new *locked* message and switch the lifelines state to *locked* state which is shown in the lower left image of Figure 6. Now, there is no possible receiver lifeline for this message so the algorithm looks for an equal free outgoing message. Since there is a free outgoing *locked* message this free outgoing message can be bound to *control* lifeline and delete the free incoming message *locked*. Hence, *control* lifeline sends the new *up* message (Figure 6 lower image). The following steps are analogue to them after adding the *down* message. After switching the *control* lifeline back to *unlocked* state there are no possible transitions with ANY-trigger and no free incoming messages left and the test case extension has finished. To check, whether the extension is correct it is checked if the final states of the lifelines of the extended test case are in a state that is in the set related to the

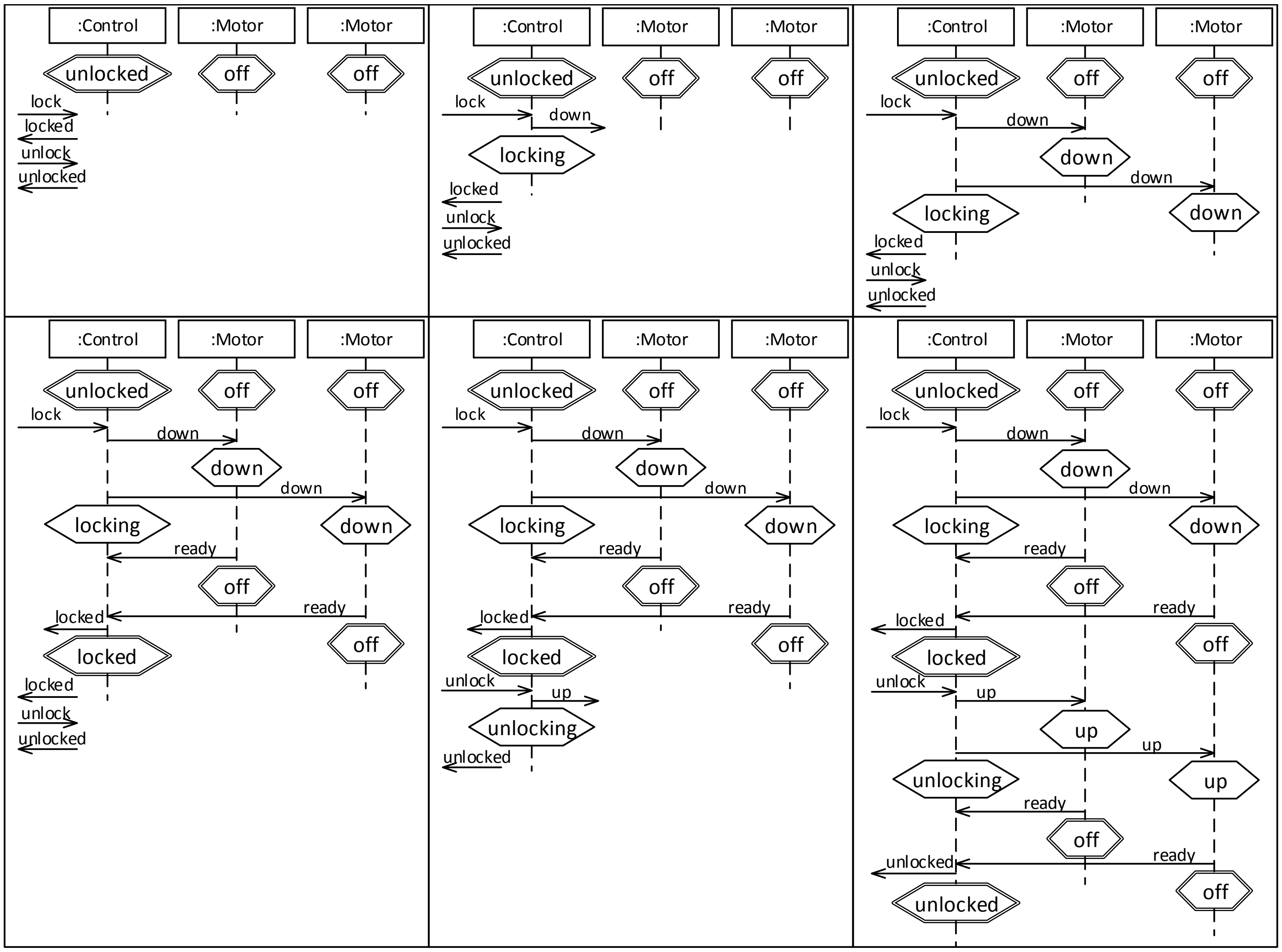

Figure 6. Extension of the sample test case

final state of the initial test case.

B. Test Case Partition

The test case that was extended by the algorithm presented in the last section has to be partitioned into several test cases for the several subcomponents. First the test case is vertically partitioned, i.e., for every lifeline only this one lifeline and only these messages, which are sent or received by this lifeline are considered. After this step there exists one test case for every lifeline that occurs in the extended test case. Now, these test cases can be partitioned horizontally at stable states, i.e., if there is a stable state that is not the initial or final state of a test case, this test case is split up there. The two new test cases contain only these messages that are sent before respectively after reaching the stable state. The stable state additionally becomes the final state of the first and the initial state of the last new test case. Since it is possible that several identical test cases are created, duplicated test cases can be refused. In our example the extended test case would be partitioned into six new test cases; two for every lifeline with each horizontal splitting at the middle stable state *locked* respectively *off*. The left image of Figure 7 shows the partitioning as dotted lines. Now, there are two times two identical test cases for the motor

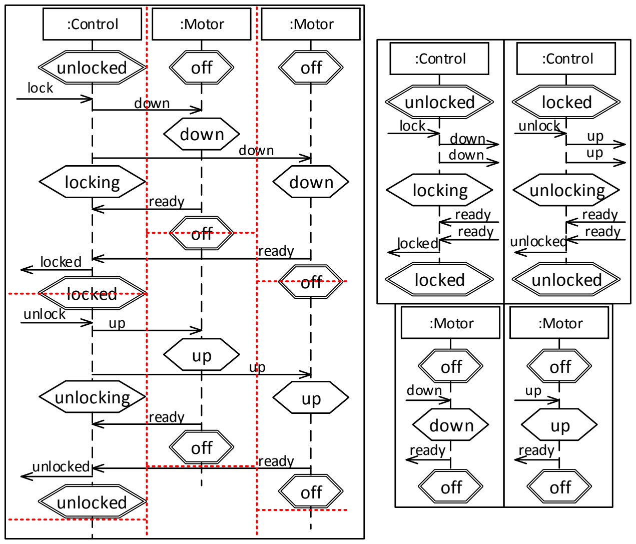

Figure 7. Partition of the extended test case

component so the duplicates can be removed and the four test cases are left that are shown in the right image of Figure 7.

C. Tool Support

To apply the techniques for describing the structure and the behaviour of the SUT as well as the test cases to test it and apply the test case decomposition a tool support will be provided. Currently the automatic test case extension based on structural and behavioural descriptions is supported. Additional graphical editors for creating the structural and behavioural descriptions and the test cases will be provided. Furthermore integration with tools for automatic test case execution and analysis like JUnit is planned.

IV. Related Work

Dias Neto et al. [11] give a survey of MBT approaches by comparing more than 400 papers. Approaches that use state based behaviour descriptions were developed amongst others from Bernard et al. [12]. They generate abstract test cases from state machines describing the behaviour of classes. Several selection criteria can be chosen. The test cases are only suitable to test classes but not components.

Another approach presented by Tretmans [13] uses labelled transition systems and the ioco-testing theory. However, the test case selection is still an open issue. The approach of Xu et al. [14] generates test cases by two different strategies: Structure-oriented and property-oriented generation. However, they also do not cover different levels of decomposition with their test cases.

The approaches of Elbaum et al [15] and Saff et al. [16] generate unit tests from system tests. But it is necessary to execute the system to get unit test. So they are not available at implementation time.

Briand et al. [17] investigated the impact of changing models on the generated test cases. They divided the test cases into three categories: Obsolete, retestable and reusable. However, they cannot update the test cases after changes, for example decompositions, of a model.

V. Conclusion and Future Work

We have presented an approach that enables an automatic test case decomposition by using the decomposition of the SUT. The test cases are extended with the systems or components internal communication and partitioned into several test cases that can be used to test the new components defined by the systems decomposition.

Future work includes consideration of the sequencing of existing and new messages during the test case extension as well as variable initial states by allowing a history for the behavioural description and the handling of indeterministic state charts. Another important aspect is tracing and impact analysis of changes of the initial test. Furthermore, the tool support will be improved by providing customized graphical editors for describing the systems structure and behaviour and the test cases and the integration of functional testing tools. After this, we plan to evaluate our approach at several existing system to compare it with existing test case generation approaches.

References

[1] M. Utting and B. Legeard, Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann, Jul. 2010.

[2] A. Pretschner and J. Philipps, “10 methodological issues in model-based testing,” in Model-Based Testing of Reactive Systems, ser. Lecture Notes in Computer Science, M. Broy, B. Jonsson, J.-P. Katoen, M. Leucker, and A. Pretschner, Eds. Springer Berlin Heidelberg, Jan. 2005, no. 3472, pp. 281–291.

[3] G. J. Myers, T. Badgett, and C. Sandler, The art of software testing. Hoboken, N.J.: John Wiley & Sons, 2004.

[4] S. Sprenkle, S. Sampath, E. Gibson, L. Pollock, and A. Souter, “An empirical comparison of test suite reduction techniques for user-session-based testing of web applications,” in Software Maintenance, 2005. ICSM’05. Proceedings of the 21st IEEE International Conference on. IEEE, 2005, pp. 587–596.

[5] M. J. Harrold, R. Gupta, and M. L. Soffa, “A methodology for controlling the size of a test suite,” ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 2, no. 3, 1993, pp. 270–285.

[6] D. Hao, L. Zhang, X. Wu, H. Mei, and G. Rothermel, “On-demand Test Suite Reduction,” in Proceedings of the 34th International Conference on Software Engineering, ser. ICSE ’12. Piscataway, NJ, USA: IEEE Press, 2012, pp. 738–748.

[7] A. Pretschner et al., “One evaluation of model-based testing and its automation,” in Proceedings of the 27th international conference on Software engineering, ser. ICSE ’05. New York, NY, USA: ACM, 2005, pp. 392–401.

[8] A. Marques, F. Ramalho, and W. L. Andrade, “Comparing model-based testing with traditional testing strategies: An empirical study,” in Software Testing, Verification and Validation Workshops (ICSTW), 2014 IEEE Seventh International Conference on. IEEE, 2014, pp. 264–273.

[9] I. Krger, R. Grosu, P. Scholz, and M. Broy, “From MSCs to statecharts,” in Distributed and Parallel Embedded Systems. Springer, 1999, pp. 61–71.

[10] C. Rupp, S. Queins, and B. Zengler, UML 2 glasklar: Praxiswissen fur die UML-Modellierung. Munchen; Wien: Hanser, 2007.

[11] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos, “A survey on model-based testing approaches: a systematic review,” in Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies: held in conjunction with the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE) 2007, ser. WEASELTech ’07. New York, NY, USA: ACM, 2007, pp. 31–36.

[12] E. Bernard et al., “Model-based testing from UML models.” in GI Jahrestagung (2), 2006, pp. 223–230.

[13] J. Tretmans, “Model based testing with labelled transition systems,” in Formal methods and testing. Springer, 2008, pp. 1–38.

[14] D. Xu, O. El-Ariss, W. Xu, and L. Wang, “Testing aspect-oriented programs with finite state machines,” Software Testing, Verification and Reliability, vol. 22, no. 4, 2012, pp. 267–293.

[15] S. Elbaum, H. N. Chin, M. B. Dwyer, and J. Dokulil, “Carving Differential Unit Test Cases from System Test Cases,” in Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ser. SIGSOFT ’06/FSE-14. New York, NY, USA: ACM, 2006, pp. 253–264.

[16] D. Saff, S. Artzi, J. H. Perkins, and M. D. Ernst, “Automatic Test Factoring for Java,” Tech. Rep. MIT-CSAIL-TR-2005-042, Jun. 2005.

[17] L. Briand, Y. Labiche, and G. Soccar, “Automating impact analysis and regression test selection based on UML designs,” in International Conference on Software Maintenance, 2002. Proceedings, 2002, pp. 252–261.