Improving Test Coverage for UML State Machines Using Transition Instrumentation

Mario Friske and Bernd-Holger Schlingloff

Fraunhofer FIRST, Kekuléstraße 7, 12489 Berlin, Germany
{mario.friske|holger.schlingloff}@first.fraunhofer.de

Abstract. We discuss the problem of generating test suites from UML state machines and present a method to extend the capabilities of existing automated test case generators. Current tools provide only a limited coverage for different testing objectives. We argue that a better coverage can be achieved by instrumenting transitions, and performing an appropriate pre- and postprocessing. We describe the necessary enhancements of the UML model and demonstrate our method on a simple example. We further report on an industrial case study where we successfully applied our method for generating a validation test suite for a safety-relevant communication protocol.

1 Introduction

Testing is one of the most time-consuming tasks in the development of complex reactive systems. Thus, it is highly desirable to obtain as much tool support as possible. In code-based testing, the tester derives test sequences from the actual program code of the implementation. Code-based testing has some major drawbacks: First, mere code-based testing cannot ensure that the observed behaviour is equivalent to the intended behaviour, second, testing can only start when an actual implementation is available, which is usually rather late in the development process, and third, whenever the implementation is modified, the test suite has to be adapted anew. In contrast, specification-based testing focuses on required properties rather than on a particular implementation. The tester regards the implementation as a black box with hidden content. The development of specification-based tests can begin as soon as there is a requirements document (i. e., even before writing the first line of code).

The effectiveness of automated specification-based test case derivation methods largely depends on the specification formalism, which is used to denote system properties. Model-based testing assumes that system properties are represented in a formal or semiformal modelling language. Often, state-based formalisms such as finite state machines [1], Statecharts [2], UML state machines [3] or Stateflow models [4] are used. These formalisms are easy to use and have a well-understood semantics. Moreover, in a model-based development process diagrams can be used to derive an implementation by stepwise refinement. In such a context, the *system model* should represent the specification (i. e., focus on

the functional requirements only). The *implementation model* is a refinement of the system model and takes implementation aspects such as data representation, efficiency, and scheduling into account.

For several modelling formalisms there exist code and test generation tools. Whereas for the generation of production code a detailed implementation model is necessary, test cases can already be generated from the system model. Various algorithms can be used for the construction of test cases. For example, one strategy is to use a breadth-first or depth-first search to generate all paths through the state graph up to a certain length. More elaborated strategies try to satisfy coverage criteria such as *All Transitions* [5] or *Modified Condition / Decision Coverage (MCDC)* [6] on the model. Even more advanced techniques use temporal logic model checking and fault injection to generate error traces, which can then be used as test sequences. A main topic in all of these approaches is to construct test suites which are both meaningful and manageable (i. e., provide a sufficient coverage and can be generated, executed and evaluated in reasonable time).

In the literature, many methods for generating test cases from state-based specifications have been proposed. Among them are methods based on FSM (e. g. [7, 8]), extended FSM (e. g. [9, 10]) or various variations of Statecharts (e. g. [11, 12]). For practical application, there are a few commercial tools (Rhapsody/ATG [13], Conformiq [14], and Reactis [15]) and a number of experimental research tools (Agedis [16], Teager [17], TGV [18], TORX [19], AGATHA [20], and others) available. While research prototypes cover a wide range of coverage criteria, commercial tools only support a very limited selection of coverage criteria.

In this paper, we present two methods for improving the test coverage of test case generators, *Transparent Transition Instrumentation* and *Extended Transition Instrumentation*. We achieve a better coverage than the originally supported criterion *MCDC* by instrumenting transitions, and performing an appropriate pre- and postprocessing. Transition Instrumentation allows us to realize additional transition-based coverage criteria (e. g., *All n-Transition Sequences* [5]) without modifying the generator. We describe the necessary enhancements of the model and demonstrate our method on a simple example. We further report on an industrial case study where we successfully applied our method for generating a validation test suite for a safety-relevant communication protocol.

This paper is structured as follows: In the next section, we discuss capabilities and limitations of currently available test case generators for UML state machines. In Sect. 3, we present the two methods *Transparent Transition Instrumentation* and *Extended Transition Instrumentation*. Section 4 describes the industrial application of these methods. Finally, in Sect. 5 we draw some conclusions and give an outlook on future research work.

2 Test Generation from UML State Machines

We use the sample state machine shown in Fig. 1 to discuss abilities and limitations of current commercial test case generators for deterministic UML state

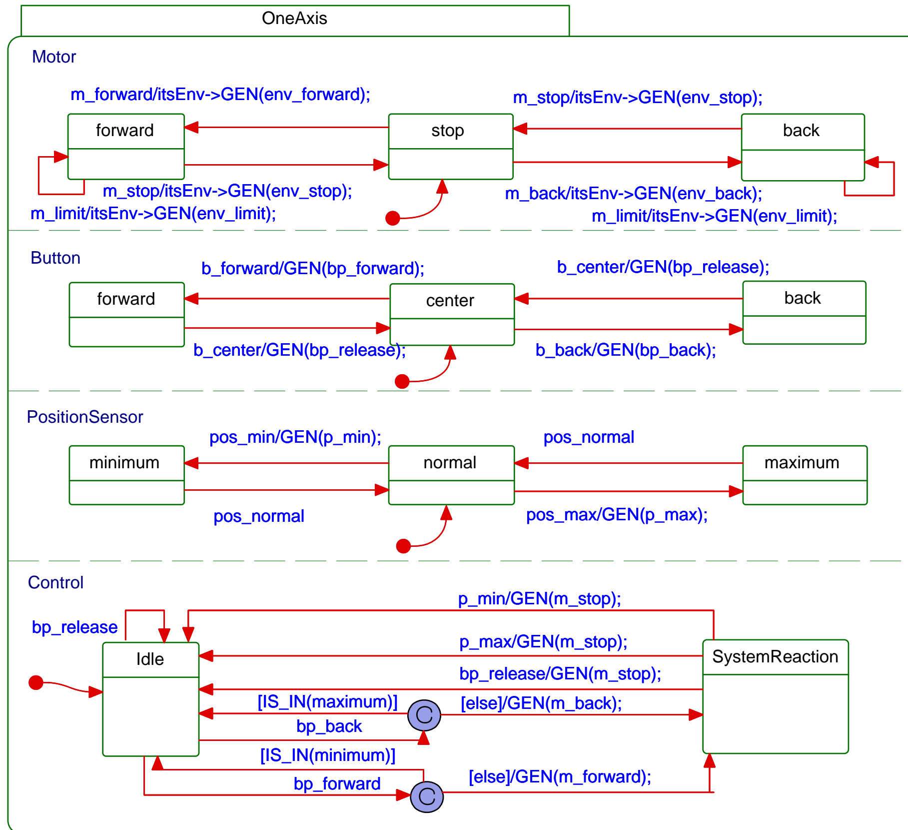

Fig. 1. State machine of simplified seat control (one axis)

machines. The state machine is a realization of one axis of the seat control specified in [21]. It allows moving a seat forward or backward by pressing a three-positions-button. When the button is pressed, the seat should move in the corresponding direction. The seat should stop moving when it reaches the final position. The seat should not move further in this direction before it has moved back into the opposite direction.

We manually created the model in a systematic approach following the modelling guidelines described in [22]. In contrast to the original guidelines, which suggest creating separate classes with their own Statechart for the control and for each sensor and actuator, we created just one class and one state machine with an orthogonal state comprising concurrent regions. As a modelling tool, we used Rhapsody in C++ [23], which utilizes C++ statements and macros as action language in UML state machines. For each sensor and each actuator, we modelled possible transitions between valid equivalence classes of input and output in a state machine. An additional state machine models the reactive be-

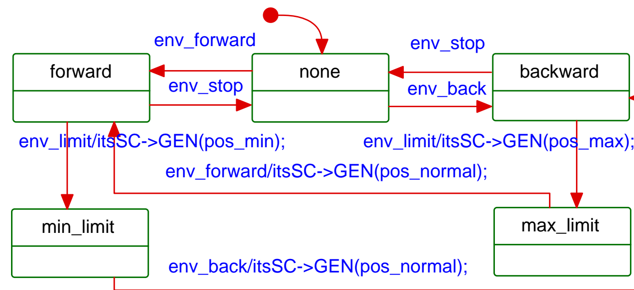

Fig. 2. Environment model of seat control

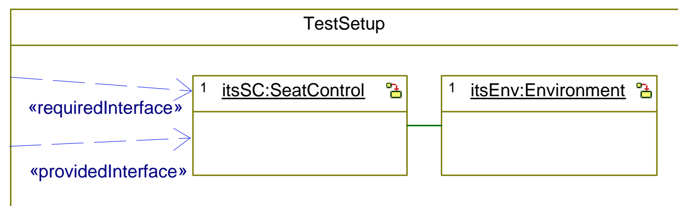

Fig. 3. Test setup including system model and environment model

haviour of the controller. The parallel composition of these four state machines (see Fig. 1) models the overall behaviour of the seat control.

An *environment model*, which describes how the environment behaves if the system is integrated into it, complements the *system model*. The environment model covers only physical feedback but does not include behaviour of the user. The state machine of the environment model displayed in Fig. 2 defines possible transitions between equivalence classes of seat movement (i. e., the feedback between motor movement and resulting positions detected by the position sensor). Whenever the motor is switched on or off, an indicating event is sent from the seat control to the environment model. There it causes a transition between states indicating the current position of the seat. Transitions in the environment model define valid changes of positions. For example, only when the seat moves backward it will reach its maximum position.

As displayed in Fig. 3, the test setup consists of one instance `itsSC` of the class `SeatControl` which communicates with an instance `itsEnv` of the class `Environment`. The state machines displayed in Fig. 1 and Fig. 2 define the behaviour of the classes `SeatControl` and `Environment`, respectively. Stereotyped dependencies specify provided interfaces (inputs) and required interfaces (outputs) for test case generation.

We used Rhapsody's event generation mechanism [23] to realize the communication between these concurrent state machines. For example, the expression `m_forward/itsEnv->GEN(env_forward)` in the upper left corner of Fig. 1 signifies that the corresponding transition fires when the trigger event `m_forward`

occurs and as a result, an event `env_forward` is generated and sent to the environment model `itsEnv`. Using such a combination of system and environment model allows modelling the interaction resulting from feedback between actuators and sensors.

For the following experiment, we use the commercially available test case generator ATG [13] to generate test cases from the combined model. The generator supports the coverage criterion *MCDC* [6] on the code generated from the state machine. *MCDC* means that every point of entry and exit in the program has been reached at least once, every condition in a decision in the program has taken on each possible outcome at least once, and each condition has been shown to affect that decision outcome independently [24]. This includes the coverage criterion *All Transitions* [5] which in turn includes the coverage criteria *All States* [5] and *All Events* [5]. *MCDC* is the generally acknowledged coverage criterion for white box testing of safety critical systems in avionics, as required by certain standards such as DO178B [24]. ATG does not explicitly handle the state machine's data space.

We convert the resulting test suite into simple sequences of events observable at the interfaces of the black box, which is the system under test. Then we free this test suite from duplicates and inclusions. Figure 4 shows the resulting test suite. It consists of six test cases which specify sequences of input events to be generated and output events to be observed. They are shown in six columns and should be read top-down.

b_back	b_forward	b_forward	b_back	b_forward	b_back
m_back	m_forward	m_forward	m_back	m_forward	m_back
m_limit	b_center	m_limit	b_center	m_limit	m_limit
m_stop	m_stop	m_stop	m_stop	m_stop	m_stop
b_center		b_center		b_center	b_center
b_forward		b_back		b_forward	b_back
m_forward		m_back			

Fig. 4. Test cases generated from Fig. 1 according to coverage criterion MCDC

Although the test suite shown in Fig. 4 satisfies the coverage criteria *MCDC*, *All States*, *All Events* and *All Transitions* in the specification, it is not sufficient for black box testing. In particular, the normal behaviour of the system is only tested in very short sequences. Many typical testing objectives for black box testing such as detecting missing or invalid states are not covered by this approach. For example, the test suite would not detect an erroneous implementation of the upper right transition with trigger event `m_stop` in Fig. 1 that leads to a different state than specified. The test suite does not contain a test case that shows the feasibility of moving the seat back again after reaching it's maximum position and subsequently moving it forward. Hence executing this test suite and observing input-output-conformance is not sufficient for ensuring that the transition reaches its target state.

Other conformance criteria and test case generation algorithms have been proposed in the literature. Amongst these are the transition tour method [25], the W-method [7] and UIO-method [8]. These methods construct test sequences to check the isomorphism or bisimilarity of finite IO-automata; that is, for each state, possible input and corresponding output of one automaton there must be an appropriate state in the other automaton which under the same input yields the same output and goes to an appropriate successor state. For black box testing these methods are only partially adequate. Firstly, they assume that both specification and implementation are given as finite automata. In our case, the specification is given as a UML state machine which may contain local variables and parameterized events, and the implementation is a black box. Secondly, these methods aim at the construction of test sequences of minimal length; during test execution, the implementation is reset after each sequence. In our case, reset is a costly operation which requires manual intervention and should be avoided. Thirdly, the goal of these methods is to construct test suites which are "complete" for some conformance criterion, such that the testing process must stop when all test cases are executed. In our case, we want to be able to adjust the coverage criterion such that the implementation can be exercised for a certain amount of time with well-defined test coverage.

In summary, for black box test generation from UML state machines the coverage which can be achieved by current commercial test case generators or other available tools is not sufficient. Hence, we are looking for ways to achieve a better coverage. Two options are apparent: One is to build a homemade test case generator. Another one is to modify a commercial test case generator.

Unfortunately, these two approaches are normally not applicable. On the one hand, it takes too much effort to build a test case generator. On the other hand, source code of commercial generators and extension APIs are mostly not available. Another aspect is that certification requires applying tools which are proven-in-use. Certification normally prevents engineers from using homemade or academic prototypes and forbids modifications of established test case generators. Hence, we have been looking for an alternative option for improving test coverage of commercial test generators. As a result, we have developed two methods that we present in the next section.

3 Improving Test Coverage

As discussed in the previous section, the coverage that can be achieved using currently available commercial test case generators is not sufficient for black box testing. Therefore, we developed an alternative approach that offers a third option. First, we present the overall approach and then we present two specific methods relying on this approach.

3.1 Extending Coverage Functions of Test Case Generators

Applying a coverage criterion to a model results in a set of test goals. A test generator generates a corresponding test suite by generating a test case for each

test goal. Test case generation might fail for some of these goals, for example resulting from unreachable code.

The basic idea of our approach is as follows: As depicted in Fig. 5, first, we enhance the state machine using a preprocessor by inserting additional elements to the model. The additional elements result in additional test goals for a test case generator leading to an enhanced coverage on the model. Then we generate test cases with an available test case generator, and finally we process the resulting test cases using a postprocessor. This allows us to generate test suites that satisfy more complex coverage criteria than the test case generator originally provides.

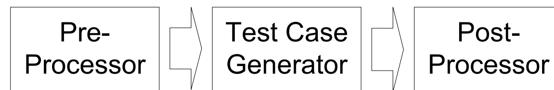

Fig. 5. Extending capabilities of test case generators through pre- and postprocessing

A test case generator provides a *generator coverage function* on a model. We augmented the input model by using an *enhancement function*. Then we generated test cases from the enhanced model. The test coverage of the resulting model is the composition of both functions, the *extended coverage function*.

In order to apply this general approach to our state-machine-based example shown in Fig. 1, we need to find a reasonable *enhancement function*. The *generator coverage function* of ATG [13] is *MCDC* on the generated code. *MCDC* includes coverage of all states, all transitions and all events in the model. Hence augmenting the state machine by including additional states, transitions, events or conditions would possibly result in larger test suites. Using additional states and transitions has the drawback of resulting in larger models while not directly leading to a larger coverage. Therefore, we did pursue the following two options for enhancing the state machine:

1. Additional variables and conditions in action code, and
2. additional events.

The first option can be used to implement counters. The main advantage of this option is transparency (i. e., no structural modification of the model is necessary and there are no extra events in the generated output). This is similar to the idea of *User Defined Test Goals* in Reactis [15], which are boolean expressions augmenting a test model.

The second option can be used to write additional information into event traces in test cases for further processing. Such information can be for instance information on the current allocation of global variables. It can also be used to pass other meta information (e. g., information about traversed paths) into test cases.

Based on these results we developed two methods for achieving extended coverage criteria: (1) *Transparent Transition Instrumentation*, resulting in test

cases, which contain longer sequences of events and comply with a given coverage criteria without changing the set of used events, and (2) *Extended Transition Instrumentation*, including additional events providing information for postprocessing. In the following subsections, we present these two methods in detail.

3.2 Transparent Transition Instrumentation

The test case generator generates test suites satisfying the coverage criterion *MCDC*. This means that for each condition in the action code two test cases are generated: in one the condition evaluates to true and in the other to false. Adding a C++-Statement to the action code of a transition that compares an additional variable with a given value results in two extra test goals for the test case generator. We can exploit this fact to implement a counter mechanism that allows sets of extra test goals resulting in generation of a desired sequence of transitions.

For a desired sequence of transitions, we use a counter variable and compare this variable with given values within condition statements and conditionally increment the variable. In other words, we add C++-Statements in the form `if(counter==n){counter++;}` to the action code of each transition in the sequence, where `n` is set according to the position of the transition within the sequence. These additional statements let a desired sequence of transitions become a goal for the test case generator. The mechanism can be exploited to force the test case generator to generate test cases that cover designated sequences of transitions and consequently to generate test suites that satisfy other test coverage criteria than *MCDC*.

Being able to generate specific sequences of transitions allows realizing all test case generation strategies relying on sequences of transitions. Sequences of transitions can be calculated based on the length of sequences as in the generation strategy *All n-Transition Sequences* [5], characterization sets as in the *W-method* [7], unique input/output sequences as in the UIO-method [8], or other criteria.

Most of these strategies based on sequences of transitions originally have been proposed for FSMs. Specific problems arise when applying these strategies to UML state machines which can be hierarchical and parallel. A strategy can be applied to either the entire state machine model with interleaved firing of transitions from all regions or to a single region. Calculation of transition sequences for complex hierarchical and parallel state machines is a non-trivial task and usually requires simulation of the state machine. Executing a transition sequence in a single region requires execution of the overall state machine and usually requires firing of several transitions in all regions.

Applying a transition-sequence-based strategy to one or more regions appears very eligible from the tester's perspective. Testers usually give more importance to certain aspects than to other aspects. Often a state machine has regions providing synchronization of other regions by communicating with them. For the previously presented state machine realizing a one-axis seat control, such central part is the region `Control` which processes inputs provided by the two

sensors (regions `Button` and `PositionSensor`) and controls the actuator (region `Motor`). The task of determining a transition sequence resulting from the overall state machine that contains the specific transition sequence in a region will be delegated to the test case generator.

In the following we explain *Transition Instrumentation* by applying the strategy *All n-Transition Sequences* [5] with $n = 2$ to the region `Control`. *All 2-Transition Sequences* requires that every specified transition sequence of length two has to be exercised at least once. The resulting test suite will achieve this coverage criterion on the region `Control` in addition to the previously achieved coverage criterion *MCDC* on the model.

Therefore, we create a more abstract alternative representation of this region with all transitions labelled using letters as shown in Fig. 6. Then we determine all sequences of length two from this representation: `ad`, `ae`, `af`, `ag`, `ah`, `bd`, `be`, `bf`, `bg`, `bh`, `cd`, `ce`, `cf`, `cg`, `ch`, `da`, `db`, `dc`, `ea`, `eb`, `ec`, `fd`, `fe`, `ff`, `fg`, `fh`, `gd`, `ge`, `gf`, `gg`, `gh`, `hd`, `he`, `hf`, `hg`, `hh`.

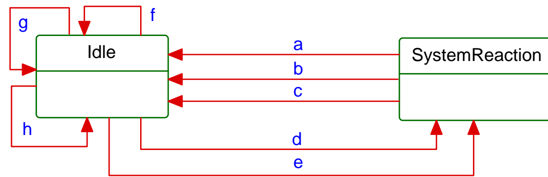

Fig. 6. Abstraction of seat control

Now we enhance the model with one counter variable of type integer for each sequence simply naming the variable after the sequence (e. g., `ad`). Then we instrument the transitions with extra code resulting in additional test goals required for achieving *MCDC*. The code is added using the following algorithm:

for each sequence of transitions
    for each transition within the sequence
        add `if(counter==n){counter++;}`

In the pseudo code `counter` stands for the integer counting variable and `n` for the number of transitions firing. For example applying this algorithm to determine the instrumentation code for exercising the transition sequence `ad` yields the following result: to the state machine presented in Fig. 1 we have to add to transition `a` the additional statement `if(ad==0){ad++;}`, and to transition `d` the statement `if(ad==1){ad++;}`.

In case that a transition has to fire twice or more, sequential order of additional statements is important, because it has to be ensured that the counter is not incremented to the maximum by a single firing. For example for achieving transition sequence `hh` the statement `if(hh==1){hh++;}` must appear before `if(hh==0){hh++;}` within the action code added to transition `h`.

After all transitions have been instrumented, generating test cases with the test case generator again will result in an extended test suite. If all transition sequences required for achieving a strategy have been coded into the model, then the resulting test suite will also fulfil the corresponding coverage criterion. The extended test suite resulting from instrumenting the state machine in Fig. 1 according to *All n-Transition Sequences* with $n = 2$ is shown in Fig. 7.

As discussed at the end of Sect. 2, the previously generated test suite displayed in Fig. 4 cannot detect erroneous implementations of the upper right transition with trigger event `m_stop` in Fig. 1. The extended set of test cases shown in Fig. 7 can detect this error (e. g., by the test case in the lower left corner).

```
b_forward
m_forward
m_limit
m_stop
b_center
b_forward
b_center
b_forward

b_back
m_back
m_limit
m_stop
b_center
b_forward
m_forward
b_center
m_stop
b_back
m_back
m_limit
m_stop

b_forward
m_forward
b_center
m_stop
b_back
m_back
m_limit
m_stop

b_back
m_back
b_center
m_stop
b_back
m_back
b_center
m_stop

b_forward
m_forward
b_center
m_stop
b_forward
m_forward

b_forward
m_forward
m_limit
m_stop
b_center
b_back
m_back
b_center
m_stop
b_forward
m_forward
m_limit
m_stop

b_back
m_back
m_limit
m_stop
b_center
b_forward
m_forward
m_limit
m_stop
b_center
b_forward

b_back
m_back
b_center
m_stop
b_forward
m_forward
m_limit
m_stop
b_center
b_forward

b_back
m_back
m_limit
m_stop
b_center
b_back
b_center
b_forward
m_forward
m_limit
m_stop
b_center
b_forward

b_forward
m_forward
m_limit
m_stop
b_center
b_forward
b_center
b_back
m_back
b_center
m_stop
b_forward
m_forward

b_back
m_back
b_center
m_stop
b_back
m_back
m_limit
m_stop
b_center
b_back

b_forward
m_forward
m_limit
m_stop
b_center
b_back
m_back
m_limit
m_stop
b_center
b_back

b_forward
m_forward
m_limit
m_stop
b_center
b_forward
b_center
b_back
m_back
m_limit
m_stop
b_center
b_back

b_back
m_back
m_limit
m_stop
b_center
b_back
b_center
b_back

b_back
m_back
m_limit
m_stop
b_center
b_back
b_center
b_forward
m_forward
b_center
m_stop
b_back
m_back
```

Fig. 7. Extended set of test cases generated from Fig. 1

3.3 Extended Transition Instrumentation

The previously explained method *Transparent Transition Instrumentation* allows to generate a set of test cases achieving a given transition-sequence-based coverage criterion. The test generator generates one test case for each goal. Thus, the number of test cases contained in the test suite can get large while each test

case is quite short. In certain situations, it is required to minimize the actual number of test cases while not reducing the coverage (i. e., to have a set of few, but long test cases). One approach to achieve this goal is concatenation of test cases. Another aspect is that depending on the model and test strategy, applying *Transparent Transition Instrumentation* for very long sequences might fail because of limited resources such as CPU-time and memory.

For example, it is possible to include all sequences of transitions of length two presented in the previous section in one test case. One possible solution is the sequence `adaeafdbdcecffgghdagfebeahfhebfhhgecgdcdbgdbhech`. This can be done by concatenating test cases individually generated for each sequence of length two.

Usually concatenation of generated test cases is more complex than simply appending the sequences displayed in Fig. 7. In a UML state machine, the current configuration of the system is not just determined by the set of current states of all sub-automata, but also by the configuration of all global variables. Hence, the concatenation of fragments cannot be done by overlapping transitions only. Such overlapping without regarding global variables might result in invalid traces. For concatenating two test cases, we must ensure that at the concatenation point the state machine has the same configuration in both test cases.

In order to take into account information about global variables during concatenation, we introduce a new event with one parameter corresponding to each global variable. We generate this event on each transition that is a potential concatenation point. If the state machine contains concurrent regions, then for each concurrent region we add one more parameter to this event. Each of these parameters represents the current state of one region as enumeration. We also introduce another type of event to record information about subsequences of events contained in this test case. The postprocessor reads both events and uses this information during test case concatenation. After the concatenation is finished, the postprocessor removes all additional events.

The sample UML state machine that we introduced in Sect. 2 does not include global variables but concurrent regions. In order to generate one sequence that covers *All n-Transition Sequences* with $n = 2$ for the region `Control`, first we generate one valid sequence[1], see above, that satisfies this criterion on the abstraction shown in Fig. 6. Then we instrument the model with counters using *Transparent Transition Instrumentation* as discussed in the previous section. Furthermore, we add the action code for generating additional events providing information about the current state of all concurrent regions.

Then we generate test cases using the test case generator and subsequently concatenate the test cases using our postprocessor. Therefore, we pass the previously generated string representing the valid target sequence as parameter to the postprocessor. We realized the concatenation by recursively selecting a test case that contains the next required transition sequence, checking if global con-

[1] Here we do not consider the initial transition leading into state `idle`. Inclusion of the initial transition would require a minimum of five test cases for achieving *All n-Transition Sequences* with $n = 2$.

figurations of the state machines match, cutting out the corresponding sequence of events, and pasting it to the tail of the concatenated test case. How to ensure matching configurations will be discussed at the end of the next section.

4 Industrial Application

We used this approach within an industrial research project to generate test cases for the slave device for an automation protocol. Two state machines, one for the master device and one for the slave device, details of which are under NDA, specified the protocol. First, we translated the slave's Statechart-like specification with pseudo action code into an executable UML state machine with C++ action code. Then, we analysed the protocol and built an additional abstraction of the state machine of the slave. The original Statechart consisted of 8 states and 15 transitions, comprising two cycles, each consisting of three states. We could create an abstraction as depicted in Fig. 8. One cycle was the normal operation cycle without occurrence of faults and the other cycle was the fault handling cycle.

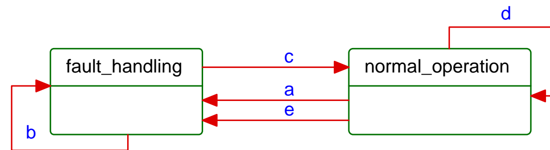

Fig. 8. Abstraction of automation protocol

For the automated generation of test cases we defined three coverage criteria:

CC_1**:** *MCDC* of the Code generated from the state machine.
CC_2**:** All sequences of length n from the abstraction of the state machine.
CC_3**:** Concatenations of m sequences obtained by CC_2 in random order.

Coverage criterion CC_1 was directly supported by the test case generator. We could achieve coverage criterion CC_2 by applying our method *Transition Instrumentation*. To this end, we calculated all possible sequences of length $n = 3$ from the abstraction in Fig. 8. We used these sequences to instrument the transitions of the state machine, as described above. Then, we generated a test suite satisfying CC_2 from it.

For achieving coverage criterion CC_3 we applied our second method *Extended Transition Instrumentation*. To this end, we extended the instrumentation for achieving CC_2 by adding extra events both for marking transitions and for writing the state of all global variables. After the test cases were generated we used our postprocessor to extract all fragments corresponding to the sequences of length $n = 3$. In order to extract fragments the postprocessor evaluated all extra events that have been added for marking sequences.

Then we calculated valid sequences of m transitions from the abstraction in random order and tried to concatenate the fragments accordingly. Comparing of global state machine configurations has shown that a simple concatenation of sequences was not always possible because of conflicting configurations. We could solve this problem by concatenation of sequences of length $n = 3$ with an overlapping of two transitions.

5 Conclusions and Further Work

We presented a method to improve the coverage capabilities of specification-based automated test generators. Whereas our considerations have been largely driven by particular application needs, there is the potential of extending them to a more abstract level.

We implemented a preprocessor which calculates sequences of transitions, determines counter variables and calculates corresponding instrumentation statements. Currently, we manually insert results of the calculations into the model. Although this is a relatively easy task, a further enhancement would be automating these steps (i. e., export the model using XMI [26], conduct an automated analysis of the model, calculate counter variables and instrumentation statements, instrument the model, reimport the model).

A further extension would be the realization of other test enhancement algorithms within our framework. We explained *Transition Instrumentation* using UML state machines, but the underlying principle is not restricted to transitions in state machines. The principle can be applied in all situations where a functional dependency between *generator coverage function*, *enhancement function*, and *extended coverage function* can be found.

A more fundamental question, which is tackled by our work, is the definition of appropriate numerical coverage criteria for specification-based testing. Currently, there are no generally acknowledged criteria for accrediting such test suites in safety-critical systems. By giving a possibility to experiment with different algorithms, our work can help in establishing such quantitative values for different safety layers.

References

1. Gill, A.: Introduction to Theory of Finite-state Machines. McGraw-Hill Education (1962)
2. Harel, D.: Statecharts: A visual formalism for complex systems. Science of Computer Programming **8**(3) (June 1987) 231–274
3. Object Management Group: Unified Modeling Language: Superstructure, version 2.0 (formal/05-07-04) (2005)
4. The Mathworks: Stateflow. http://www.mathworks.com/products/stateflow/
5. Binder, R.V.: Testing Object-Oriented Systems: Models, Patterns, and Tools. Object Technology Series. Addison Wesley (1999)

6. Hayhurst, K.J., Veerhusen, D.S., Chilenski, J.J., Rierson, L.K.: A practical tutorial on modified condition/decision coverage. Technical Report NASA/TM-2001-210876, NASA (2001)
7. Chow, T.: Testing software design modeled by finite-state machines. IEEE Transactions on Software Engineering **SE-4** (May 1978) 178–187
8. Sabnani, K., Dahbura, A.: A protocol test generation procedure. Computer Networks and ISDN Systems **15**(4) (1988) 285–297
9. Wang, C.J., Liu, M.T.: Generating test cases for EFSM with given fault models. In: INFOCOM. (1993) 774–781
10. Petrenko, A., Boroday, S., Groz, R.: Confirming configurations in EFSM testing. IEEE Trans. Softw. Eng. **30**(1) (2004) 29–42
11. Hong, H.S., Kim, Y.G., Cha, S.D., Bae, D.H., Ural, H.: A test sequence selection method for statecharts. Software Testing, Verification and Reliability **10**(4) (2000) 203–227
12. Gnesi, S., Latella, D., Massink, M.: Formal test-case generation for UML statecharts. In: ICECCS '04: Proceedings of the Ninth IEEE International Conference on Engineering Complex Computer Systems Navigating Complexity in the e-Engineering Age (ICECCS'04), Washington, DC, USA, IEEE Computer Society (2004) 75–84
13. I-Logix: Rhapsody Automatic Test Generator, Release 2.3, User Guide. (2004)
14. Conformiq Software Ltd.: Conformiq test generator. http://www.conformiq.com/
15. Reactive Systems Inc.: Reactis. http://www.reactive-systems.com/
16. Hartman, A.: Agedis final project report. Technical report, AGEDIS Consortium (2004)
17. Santen, T., Seifert, D.: TEAGER - test automation for UML state machines. In Biel, B., Book, M., Gruhn, V., eds.: Software Engineering. Volume 79 of LNI., GI (2006) 73–84
18. Fernandez, J.C., Jard, C., Jéron, T., Viho, C.: Using on-the-fly verification techniques for the generation of test suites. In: CAV '96: Proceedings of the 8th International Conference on Computer Aided Verification, London, UK, Springer (1996) 348–359
19. Tretmans, J., Brinksma, E.: Côte de resyste – automated model based testing. In Schweizer, M., ed.: Progress 2002 – 3rd Workshop on Embedded Systems, Utrecht, The Netherlands, STW Technology Foundation (October 24 2002) 246–255
20. D. Lugato and C. Bigot and Y. Valot and J.P. Gallois and S. Gerard and F. Terrier: Validation and automatic test generation on UML models : the AGATHA approach. Journal of Software Technology Transfer (2004)
21. Houdek, F., Paech, B.: Das Türsteuergerät - eine Beispielspezifikation. IESE-Report Nr. 002.02/D, Fraunhofer IESE (2002)
22. Denger, C., Kerkow, D., von Knethen, A., Medina Mora, M., Paech, B.: Richtlinien - Von Use Cases zu Statecharts in 7 Schritten. IESE-Report Nr. 086.02/D, Fraunhofer IESE (2002)
23. I-Logix: Rhapsody in C++, Version 6.0, User Guide. (2004)
24. RTCA: DO-178B, Software considerations in airborne systems and equipment certification (1992)
25. Naito, S., Tsunoyama, M.: Fault detection for sequential machines by transition tours. In: Proccedings of the 11th. IEEE Fault Tolerant Computing Symposium. (1981) 238–243
26. Object Management Group: XML Metadata Interchange (XMI) Specification. OMG, http://www.omg.com/ (2003)