Combining Structural and Functional Test Case Generation

Christian Pfaller, Markus Pister

Lehrstuhl Informatik IV – Software & Systems Engineering
Technische Universität München, 85748 Garching
pfaller, pister@in.tum.de

Abstract: Model-based testing uses a test model which defines in general an infinite set of correct system runs. The task for the test case generator is to select an "interesting" subset of all the traces. The way this subset is chosen is defined by the *test case specification*. Two types are widely used: *structural* or *functional* test case specifications. The first is given in terms over the model elements, the later is derived from the underlying system requirements. Whereas structural criteria are easily applicable to any model and support fully automation, functional specifications need more manual definition but are closer to the users' requirements.

In this paper we propose a method which combines advantages of both: structural criteria and functional test case specification. Especially it supports automation in large parts but focuses on the users' requirements as well. Furthermore the method provides an easy and flexible adjustment to project-specific needs. By setting up parameters the tester is able to influence the functional focus of the generated tests.

Key ideas of the method are a classification and weighting of requirements and the selection of test cases only from parts of the original model: For every requirement a sub-model is selected that is defined by the requirement and the weighting of the requirements' class.

1 Introduction

Model-based test case generation has a high potential to significantly reduce the costs of testing activities in software development. In the field of (embedded) reactive systems, where a model of the system under test (SUT) can be described by a state machine, model-based test case generation has already shown its practical feasibility, for example in [Pr05]. A model of the SUT is used to retrieve a set of test cases according to a *test case specification*. In most cases therefore structural coverage criteria are used, as in the techniques described in [HMR04, Pr04]. Reactive systems process a potentially infinite sequence of inputs. Thus the set of all possible input sequences is infinite. The generation of test cases means to select "interesting" test cases from the set of all possible sequences. A *test case specification* is used to define the test cases to be selected. In general the model and the test case specification may be regarded as the inputs of a test case generator. Test case specifications may be *structural*, *stochastic* or *functional*. Structural test case specifications use coverage criteria like state coverage, condition coverage, or MC/DC coverage [Pr03, RH03]. Their advantage is the support for fully automation and the consideration of the whole model for testing. However, structural criteria do not refer to the requirements and provide hardly any flexibility for specific adjustments in projects. Functional test case specifications, which are manually defined according to the requirements can be adjusted to specific projects. Their drawback is the high effort for formulation and the danger of not considering the interdependencies between requirements which occur in the model.

1.1 Problem and Contribution

To ensure a certain level of software quality the various stakeholders—especially the clients—usually require to provide suitable tests for every single requirement they had stated. A trivial fulfillment of ”one test case per requirement” should be avoided, since crucial execution paths resulting from (not explicitly stated) combinations of requirements will likely be omitted. The manual definition of test cases is usually a laborious task, also when models are used for test case generation but functional test case specification must be defined. On the other hand fully automated test case generation which exploit just the structure of a model is hardly to adjust to requirements and project specific settings.

In this paper we propose a method which supports a high degree of automated test case generation that is guided by the explicitly defined users' requirements. The test case generation is easily adjustable to the project specific settings and traceability from test cases to the original requirements is given. Furthermore the proposed method pays special attention for testing interdependencies between different requirements and has the ability to distinguish between requirements which should be focused more or less during testing. Finally the test method leads to fulfill a higher requirements coverage.

1.2 Related Work

Test case generation by means of structural coverage criteria is described in [HMR04, Pr03, Pr04, RH03]. Unlike these approaches the presented work not only bases on the structure of a test model or test object respectively but mainly on the specified requirements. In [GHN93] test cases are derived directly from message sequence charts. In contrast, this work focuses on the integration afore mentioned model-based generation techniques together with a requirements-based test case specification. In [GH99] test cases are generated from SCR specifications but in contrast to the presented work here this approach requires that every detail of functionality of interest is explicitly stated as a requirement. In [OB88] category partition is proposed, a systematic way to define test scripts based on equivalence classes which are derived from requirements. A comparison of many requirements based testing methods was done in [DM03], none of these uses a model were all requirements are integrated. Similar to [HDW04] we see the need for coverage criteria based on requirements. To the authors knowledge not much work is done yet on requirements-based coverage criteria for model based testing. In [Pf06] a development method is outlined in which requirements (there called *services*) shall be used as test case

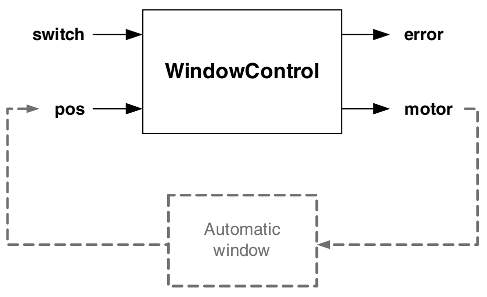

Figure 1: Interfaces of *WindowControl* example

specification.

1.3 Outline

In the next section we introduce a running example which will be used throughout the paper. In section 3 the specification of requirements is described whereas section 4 addresses the completion of the specification by building the behavior model. The test case generation method is presented in section 5 and section 6 compares a classical test method against results of our new method. Finally we state our conclusion and outline future work.

2 Running Example

As a running example we use a simple control system for an automatic window of a car. Figure 1 illustrates the interfaces of *WindowControl* to its environment.

We assume that the system boundaries as well as the input and output ports of the system were already identified: *WindowControl* has two inputs, the *switch* signal where the command for opening or closing the window occurs and the *pos* signal which signals the position of the window (at top, bottom or somewhere in between) or indicates if the window is currently moving. The outputs are: *motor* which controls the motor of the automatic window and *error* which signals a possible error for logging in the diagnostics memory of the car. The possible signals at these ports are defined by the corresponding value set, $\perp$ is used to indicate the absence of the respective signal:

$$\begin{aligned} SWITCH &:= \{open, close, \perp\} \\ POS &:= \{top, bottom, middle, moving\} \\ MOTOR &:= \{down, up, stop\} \\ ERROR &:= \{err, \perp\} \end{aligned}$$

The example bases on the window control system, described in the specification [HP02], which is a representative for requirements document in the automotive industry today.

3 Analyzing the Requirements

3.1 Informal Requirements

Even though there are many precise notions—like temporal logic—the industry still uses informal, textual descriptions for requirements specification. The challenge is to bridge the gap between developers and designers prefering a more formal specification and the various stakeholders of a system only understanding the informal (and often imprecise) statements. For the *WindowControl* example we can identify the following informal requirements:

- If the switch is in position open, the window moves down (**R1**). The movement stops if
  – no signal from the switch is sent anymore (**R2**)
  – or the window is in the bottom position (**R3**)
  – or the switch signal occurs for opening the window (**R4**)
  – or the window does not send the movement signal anymore, in this case an error will be logged (**R5**)
- If the switch is in position up, the window moves up (**R6**). The movement stops if
  – no signal from the switch is sent anymore (**R7**)
  – or the window is in the top position (**R8**)
  – or the switch signal occurs for moving the window down (**R9**)
  – or the window does not send the movement signal anymore, in this case an error will be logged (**R10**)

Note that these requirements are not complete: For example they do not state what should happen if the switch is pressed to open the window but the window is already fully open. Also the original requirements document [HP02] does not describe the situation.

3.2 Formal Scenario Specification

Informal requirements are in most cases the only way to communicate a specification to all stakeholders. We need a formal representation of the requirements to support more automation in requirements based testing, too. The connection between informal and formal representation of a requirement should be easily visible for tracing and validating purposes. Thus the aim is to use a simple language which allows non-computer-scientist and non-mathematicians to validate the correctness of the formal specification. To achieve this we use input/output scenarios which state typical example system runs for every requirement. Each scenario is just a sequence of pairs of inputs and outputs at the systems' interface. (Here input or output action means the full assignment of values to all input or output ports of the system. Thus an action is usually a vector of simple values.) Formally for a set of input actions I, a set of output actions O and $n \in I\!N$ with $n \geq 0$, the set

$$TRACES_{I,O} = \{\langle (i_0, o_0), \ldots (i_n, o_n) \rangle \mid \forall k, 0 \leq k \leq n : \; i_k \in I \wedge o_k \in O\}$$

denotes the set of all possible sequences over the input alphabet I and the output alphabet O. The *WindowControl* example holds $i_k \in SWITCH \times POS$ and $o_k \in MOTOR \times ERROR$. By a finite set $REQ \subset TRACES$ the set of requirement scenarios is denoted. Every $req \in REQ$ states an example of a *typical* and *correct* system run for a informal stated functional requirement. In the example above a formal scenario for requirement **R1** could be:

$$R1 : \left\langle \underbrace{\left(\begin{pmatrix} open \\ middle \end{pmatrix}, \begin{pmatrix} down \\ \perp \end{pmatrix} \right)}_{step0}, \underbrace{\left(\begin{pmatrix} open \\ moving \end{pmatrix}, \begin{pmatrix} down \\ \perp \end{pmatrix} \right)}_{step1} \right\rangle$$

3.3 Pre-Conditon

The notation of scenarios assumes that every scenario starts in a defined initial state of the system. Usually not all requirements are described from the initial state on and only hold if some specific actions have been executed before. For example the *WindowControl* requirements **R2** – **R5** assume that the window is moving down. Thus the actions performed to move the window down are called the *pre-condition* of the requirement and a scenario for a requirement consists of a (possibly empty) pre-condition and a main part. Only the main part of a scenario reflects the rationale behind a requirement. In most cases the pre-condition is defined as a separate requirement on its own. In table 1 the scenarios for the requirements considered are stated. There actions in brackets denote the pre-condition.

3.4 Classification of Requirements

The proposed method for requirements-based testing allows to consider different classes of requirements where requirements of one class shall be more important for testing then others. There may be several reasons for classifying requirements in such a way, for example: Safety-critical requirements may be concerned in more depth; for a new variant of a system the testers interest may lie mainly in the specific requirements for that variant; requirements which are more likely to be executed by the users should be reflected in more tests; or requirements which were changed or introduced during development and were not part of the initial requirements documents should be tested more extensively, since these may be not that elaborated than other requirements.

In Table 1 scenarios for all informal requirements of *WindowControl* are stated. The original requirements **R4** and **R9** are replaced by **R4a / R4b** and **R9a / R9b**. In our example

		step	0	1	2	3
R1	In	switch	open	open		
		pos	middle	moving		
	Out	motor	down	down		
		error	$\perp$	$\perp$		
R2	In	switch	(open)	(open)	$\perp$	
		pos	(middle)	(moving)	moving	
	Out	motor	(down)	(down)	stop	
		error	($\perp$)	($\perp$)	$\perp$	
R3	In	switch	(open)	(open)	open	
		pos	(middle)	(moving)	bottom	
	Out	motor	(down)	(down)	stop	
		error	($\perp$)	($\perp$)	$\perp$	
R4a	In	switch	(open)	close	close	close
		pos	($\perp$)	moving	$\perp$	$\perp$
	Out	motor	(down)	$\perp$	$\perp$	$\perp$
		error	($\perp$)	$\perp$	$\perp$	$\perp$
R4b	In	switch	(open)	close	$\perp$	close
		pos	($\perp$)	moving	$\perp$	$\perp$
	Out	motor	(down)	$\perp$	$\perp$	up
		error	($\perp$)	$\perp$	$\perp$	$\perp$
R5	In	switch	(open)	(open)	open	
		pos	(middle)	(moving)	middle	
	Out	motor	(down)	(down)	stop	
		error	($\perp$)	($\perp$)	err	

		step	0	1	2	3
R6	In	switch	close	close		
		pos	middle	moving		
	Out	motor	up	up		
		error	$\perp$	$\perp$		
R7	In	switch	(close)	(close)	$\perp$	
		pos	(middle)	(moving)	moving	
	Out	motor	(up)	(up)	stop	
		error	($\perp$)	($\perp$)	$\perp$	
R8	In	switch	(close)	(close)	close	
		pos	(middle)	(moving)	top	
	Out	motor	(up)	(up)	stop	
		error	($\perp$)	($\perp$)	$\perp$	
R9a	In	switch	(close)	open	open	open
		pos	($\perp$)	moving	$\perp$	$\perp$
	Out	motor	(up)	$\perp$	$\perp$	$\perp$
		error	($\perp$)	$\perp$	$\perp$	$\perp$
R9b	In	switch	(close)	open	$\perp$	open
		pos	($\perp$)	moving	$\perp$	$\perp$
	Out	motor	(up)	$\perp$	$\perp$	down
		error	($\perp$)	$\perp$	$\perp$	$\perp$
R10	In	switch	(close)	(close)	close	
		pos	(middle)	(moving)	middle	
	Out	motor	(up)	(up)	stop	
		error	($\perp$)	($\perp$)	err	

Table 1: Example scenarios for the informal stated requirements. (Actions in brackets denote the pre-condition)

we assume the following additional requirement which has been revealed later (e. g. due to some technical limitations of the switch): ”If the switch switches from open to close (or vice versa) the movement of the window must stop until the switch was fully released, thus no signal from the switch is sent.”

Thus **R4** and **R9** had to be adapted. We are now, for example, interested in test cases which focus specially on the new added requirements. Hence we classify the requirements in the set REQ_{orig} of original requirements and in the set REQ_{new} of new added requirements:

$$\begin{aligned} REQ_{orig} &= \{\mathbf{R1}, \mathbf{R2}, \mathbf{R3}, \mathbf{R5}, \mathbf{R6}, \mathbf{R7}, \mathbf{R8}, \mathbf{R10}\} \\ REQ_{new} &= \{\mathbf{R4a}, \mathbf{R4b}, \mathbf{R9a}, \mathbf{R9b}\} \end{aligned}$$

4 Building the Model

The scenario specifications as well as the informal requirements serve as the basis for building the behavior model of the system. The model should be given in some state machine or in the form of some state charts [Ha87] dialect.

4.1 Completion of the Specification

In the proposed test generation method two properties must hold for the model: (a) the input and output alphabet of the model and the set of scenarios must be equal or a bijection between the alphabets must be defined. (b) The model must accept all defined requirement scenarios. Since the scenarios—and the informal requirements, too—do in general not specify the complete behavior of the system, additional information is added during modeling. In contrast to the scenarios the model is total and deterministic, thus for every input sequence the corresponding output sequence has to be retrieved. The additional information contained in the model may stem from implicit or domain-specific knowledge or according to the experience of the designer (see the note in end of section 3.1).

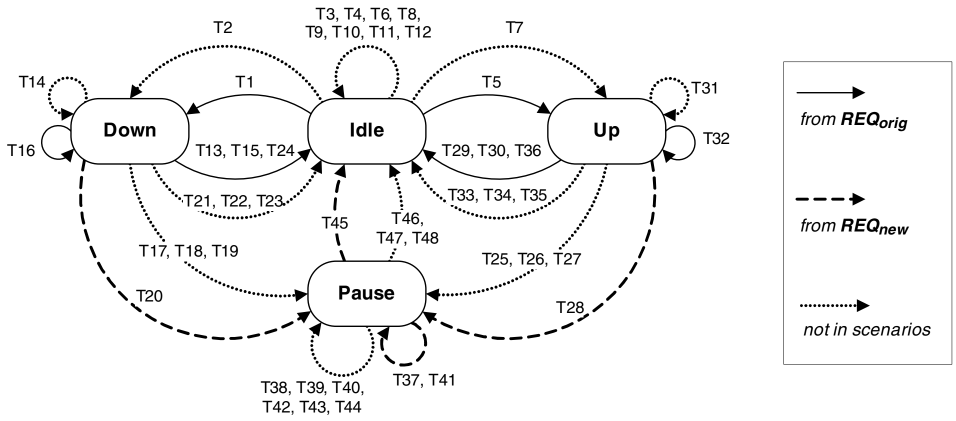

Figure 2: Model of *WindowControl*. Each edge denotes a set of transitions identified by Tx; see table 2 for details. The different styles of the edges represent the class of the transitions.

4.2 Classification of Transitions

Figure 2 and table 2 specify a model of *WindowControl* given as Mealy machine [Me55]. The classification of requirements introduced in section 3.4 is assigned to the transitions in the model: If a transition is fired during the execution of the main part of scenario r the class of the transition will be the class of r. (If scenarios of different classes fire the same transition a prioritization of classes must be defined.) For transitions which are not contained in any scenario no class is assigned. Note that in our example the transitions which lead from/to the *Pause* state are added due to the later introduced requirements REQ_{new}.

tr	Soure	Input		Output		Dest.	fired by	Class
	State	switch	pos	motor	error	State	scenario	
T1	Idle	open	⊥	down	⊥	Down	R1	orig
T2	Idle	open	top	down	⊥	Down		–
T3	Idle	open	bottom	⊥	⊥	Idle		–
T4	Idle	open	moving	⊥	err	Idle		–
T5	Idle	close	⊥	up	⊥	Up	R6	orig
				⋮				
T13	Down	open	⊥	⊥	err	Idle	R5	orig
T14	Down	open	top	down	⊥	Down		–
T15	Down	open	bottom	⊥	⊥	Idle	R3	orig
T16	Down	open	moving	down	⊥	Down	R1	orig
				⋮				
T20	Down	close	moving	⊥	⊥	Pause	R4a, R4b	new
				⋮				
T24	Down	⊥	moving	⊥	⊥	Idle	R2	orig
				⋮				
T28	Up	open	moving	⊥	⊥	Pause	R9a, R9b	new
T29	Up	close	⊥	⊥	err	Idle	R10	orig
T30	Up	close	top	⊥	⊥	Idle	R8	orig
T31	Up	close	bottom	up	⊥	Up		–
T32	Up	close	moving	up	⊥	Up	R6	orig
				⋮				
T36	Up	⊥	moving	⊥	⊥	Idle	R7	orig
T37	Pause	open	⊥	⊥	⊥	Pause	R9a	new
				⋮				
T41	Pause	close	⊥	⊥	⊥	Pause	R4a	new
				⋮				
T45	Pause	⊥	⊥	⊥	⊥	Idle	R4b, R9b	new
T46	Pause	⊥	top	⊥	⊥	Idle		–
T47	Pause	⊥	bottom	⊥	⊥	Idle		–
T48	Pause	⊥	moving	⊥	err	Idle		–

Table 2: Excerpt of the transition relation for the Mealy machine in the *WindowControl* example. Also the scenarios in which a transition is fired is stated as well as the resulting class of the transition. (In the table we left out some of the transition where no class could be assigned.)

5 Test Case Generation

Since we want to get test cases according to the specified requirements (or its scenarios respectively) the test case generation is done separately for each requirement. The full set of test cases for the system is the union of test cases generated for each requirement. The key idea of the method is to retrieve a specific sub-model for every requirement and to generate test cases from these sub-models.

5.1 Building the Sub-Model for a Requirement Scenario

The sub-model for a specific requirement is built according to the classification of the transitions and a weighting of the classes. For k different classes the weighting is defined by a vector $(l_0, l_1, ..., l_k)$ with every $l_i \in I\!N, l_i \geq 0$. Here l_0 is the default weighting for transitions which are in no class and $l_i, 0 > i \geq k$ is the weighting for the k different classes. For the *WindowControl* example with classes $orig$ and new let us choose the weighting $(l_0, l_{orig}, l_{new}) = (0, 1, 2)$.

The algorithm for selecting the sub-model is stated in figure 3. Consider for example requirement **R1** of *WindowControl*: In the main part of **R1** transitions **T1** and **T16** fire, the only state reached by these transitions is **Down**. With the above defined weighting $(0, 1, 2)$ in addition all paths starting in **Down** are selected which comprise a maximum of **one** transition in class REQ_{orig} and a maximum of **two** transitions in REQ_{new} (and no unclassified transitions[1]). A possible path is for example **T20–T45–T5**. In figure 4 the resulting sub-model for **R1** is shown as well as the sub-model for **R4a**.

```
For a requirement scenario r and the weighting (l0, l1, ... lk)
    select the set Trans(r) of transitions which fire in the main part of r;
    add Trans(r) to the sub-model;
    from every state s which is reached by a transition in Trans(r)
        search for all paths p starting in s where
            p comprises a maximum of li transitions of class i;
            add the transitions on p to the sub-model;
    add all states connected to one of the transitions in the sub -model;
```

Figure 3: Algorithm for building the sub-models

The test case generation is applied to every requirements-based sub-model. A structural test case specification, like transition coverage, is used to support automation. Thus a tracing is given from a requirement to the set of test cases generated from the requirements-specific sub-model.

5.2 Choosing the right Weighting

By applying the algorithm of the recent section, sub-models are built according to the weighting of requirement classes. The effect of a higher weighting of a requirements class is a stronger influence on the sub-models for *all* requirements—thus the requirements in the class are tested in more detail in the combination with all other requirements. For the question on an optimal setting of the weighting no general answer can be stated because it highly depends on the project specific properties: Size of the model, distance of the model to a complete graph and the size of the set of very detailed requirements scenarios. Two situations should be avoided: Setting the weightings too high the sub-model may

[1]Of course, in reality it is not wise to set one of the weighting parameters to 0 since this will exclude these transitions in the sub-model. But in the small *WindowControl* example used in this paper we set $l_0 = 0$ because otherwise building the sub-models would result in the full model again. This problem should not arise in realistic models which are quite larger in general.

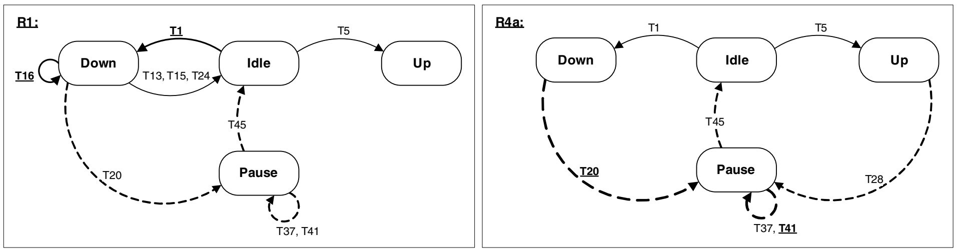

Figure 4: Examples of sub-models for *WindowControl* requirements **R1** and **R4a**. (Transitions which are contained in the requirement are underlined. Notion of transitions is in Figure 2)

often result in the complete model again, setting the weightings too low the resulting sub-models may only comprise the original requirements scenario but not much more.

In contrast to only use structural criteria for test case generation the proposed method has the advantage that by the weighting parameters the test case generation can be adjusted to the specific situation in a project. We believe that most experienced testers have an intuitive understanding what ”valuable” test cases would look like. Fully-automated test generation approaches may indeed often result in test cases which fulfill a certain coverage criteria but it is also very likely that such test cases look somehow odd in the eye of an experienced tester. He may have chosen a very different set of test cases manually. By applying our method the tester has the possibility to adjust the weighting parameters and direct the test case generation to achieve appropriate results. An iterative process could be applied: The weighting will be set to some initial values first and the resulting test cases are reviewed. The weighting then may be adjusted if necessary.

6 Example Results

We have applied the proposed method which uses requirements-based sub-models as well as test case generation on the full model to the *WindowControl* Example. As well as in the sub-models as in the full model we used transition coverage and the CLP-based test case generator of *AutoFOCUS* [Pr04].

When generating tests from the full model—which is here the union of the sub-models[2]—we retrieved by transition coverage 12 test cases with an average length of 2.25 steps. Transition **T1** was executed in 5 test cases, **T5** in 7, **T28** in 3 and all others in one test case.

Generating tests form the sub-models resulted first in a total of 62 test cases, between 4 and 8 test cases from every requirements' sub-model. After removing duplicates and test

[2]The original model of figure 2 contains many unclassified transitions which are not contained in any of the later created sub-models due to the $(0, 1, 2)$ weighting we chose. For making results more comparable we used for test case generation of the full model instead the model which results from the union of submodes, which does not contain the unclassified transitions

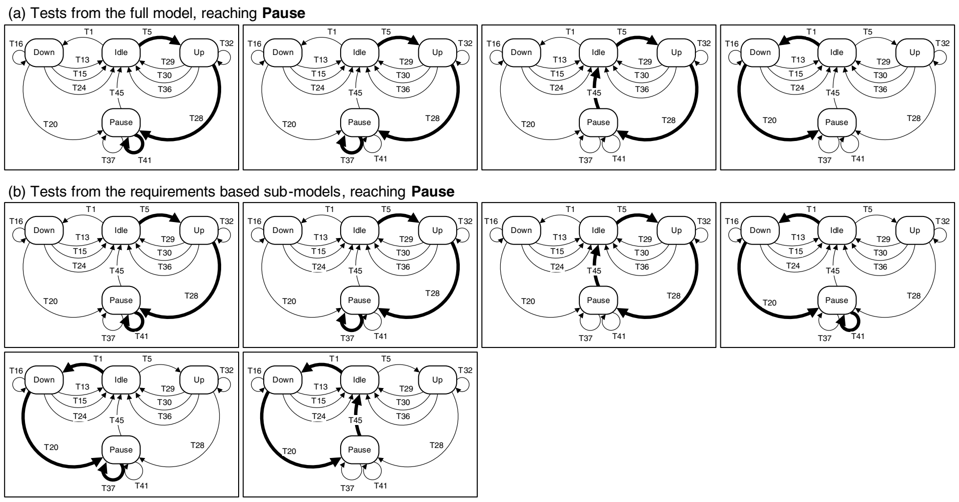
(a) Tests from the full model, reaching **Pause**
(b) Tests from the requirements based sub-models, reaching **Pause**
Down
Idle
Up
Pause
T1
T5
T13
T15
T16
T20
T24
T28
T29
T30
T32
T36
T37
T41
T45

Figure 5: Comparision of test cases reaching the *Pause* state. (Test cases are indicated by the sequence of bold transitions in the model.)

cases which were prefixes of other test cases the set of test cases comprised finally 14 test cases with an average length of 2.43 steps. Transitions **T1** and **T5** were each contained in 7 test cases, **T20** and **T28** in 3, **T37**, **T41** and **T45** in 2 and all other transitions in one test case.

The later added requirements introduced in section 4 of REQ_{new} resulted in adding the **Pause** state. In figure 5 we illustrate the test cases generated from (a) the full model and (b) from the requirements specific sub-models reaching this state. Tests from the full model using transition coverage result in 4 test cases whereas tests from the sub-models result 6 test-cases. By using the full model only one test case (which is also shorter) was generated which tests requirements of REQ_{new} in combination with the function of moving the window down. Using test generation from the sub-models results in contrast in three test cases which combine moving the window down with the requirements of REQ_{new} as well as in three test cases where these requirements are combined with moving the window up. This was the effect of the higher weighting of REQ_{new}.

7 Conclusion and Further Work

We have proposed a test method for generating requirements based tests. The method combines the advantages of structural and functional test case specifications: It supports automation in large parts, is based on the explicit defined requirements of the system, is adjustable to the projects' needs and improves testing of combinations of requirements. The method requires scenarios as examples for each requirement. A classification and weighting of requirements allows to control the influence of requirements on the resulting test cases. The tester has the possibility to control the generation process by adjusting weighting parameters. The results of the example show that an improved coverage of the requirements can be achieved.

Further work will include larger case studies as well as describing the method for more elaborated requirement specification techniques (like message sequence charts) and other notions of state machines, e. g. state charts. We assume that these are quite straightforward tasks and expect that the principles of the method will hold as well. Furthermore we are working on a metrics for quantifying requirements coverage; first results indicate that we are able to measure a better requirements coverage with the proposed method.

References

[DM03] C. Denger and M. Medina Mora. Test case derived from requirement specification. IESE-Report 033.03/E, Fraunhofer IESE, Kaiserslautern, Germany, April 2003.

[GH99] Angelo Gargantini and Constance Heitmeyer. Using Model Checking to Generate Tests from Requirements Specifications. *Software Engineering - ESEC/FSE'99: 7th European Software Engineering Conference, Held Jointly with the 7th ACM SIGSOFT Symposium on the Foundations of Software Engineering, Toulouse, France, September 1999. Proceedings*, pages 146–162, 1999.

[GHN93] Jens Grabowski, Dieter Hogrefe, and Robert Nahm. Test Case Generation with Test Purpose Specification by MSCs. In O. Faergemand and A. Sarma, editors, *SDL'93 - Using Objects*, North-Holland, October 1993.

[Ha87] David Harel. Statecharts: A Visual Formulation for Complex Systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[HMR04] G. Hamon, L. de Moura, and J. Rushby. Generating efficient test sets with a model checker. In *Software Engineering and Formal Methods, 2004. SEFM 2004. Proceedings of the Second International Conference on*, pages 261–270, 2004.

[HDW04] M.P.E. Heimdahl, G. Devaraj, and R. Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *High Assurance Systems Engineering, 2004. Proceedings. Eighth IEEE International Symposium on*, pages 178–186, 2004.

[HP02] Frank Houdek and Barbara Paech. Das Türsteuergerät eine Beispielspezifikation. IESE-Report 002.02/D, Fraunhofer Institut Experimenteles Software Engineering (IESE), Kaiserslautern, Germany, January 2002.

[Me55] G. H. Mealy. A method for Synthesizing Sequential Circuits. *Bell System Technical Journal*, 34(5):1045 – 1079, September 1955.

[OB88] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating fuctional tests. *Commun. ACM*, 31(6):676–686, 1988.

[Pf06] Christian Pfaller, Andreas Fleischmann, Judith Hartmann, Martin Rappl, Sabine Rittmann, and Doris Wild. On the integration of design and test: a model-based approach for embedded systems. In *AST '06: Proceedings of the 2006 international workshop on Automation of software test*, pages 15–21, New York, NY, USA, 2006. ACM Press.

[Pr05] Alexander Pretschner, Wolfgang Prenninger, Stefan Wagner, Christian Kühnel, Martin Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05: Proceedings of the 27th international conference on Software engineering*, pages 392–401, 2005.

[Pr03] Alexander Pretschner. *Zum modellbasierten funktionalen Test reaktiver Systeme*. PhD thesis, Technische Universität München, Fakultät für Informatik, 2003.

[Pr04] A. Pretschner, O. Slotosch, E. Aiglstorfer, and S. Kriebel. Model-based testing for real. *International Journal on Software Tools for Technology Transfer (STTT)*, 5(2 - 3):140–157, March 2004.

[RH03] S. Rayadurgam and M.P.E. Heimdahl. Generating MC/DC adequate test sequences through model checking. In *Software Engineering Workshop, 2003. Proceedings. 28th Annual NASA Goddard*, pages 91–96, 2003.