Test Sequence Generation from Classification Trees using Multi-agent Systems

Peter M. Kruse

Berner & Mattner Systemtechnik GmbH,
Gutenbergstr. 15, 10587 Berlin, Germany
peter.kruse@berner-mattner.com
http://www.berner-mattner.com

Abstract. The combinatorial test design and combinatorial interaction testing are well studied topics. For the generation of dynamic test sequences from a formal specification of combinatorial problems, there has not been much work yet. The classification tree method implements aspects from the field of combinatorial testing. We will extend the classification tree by additional information to allow the interpretation of the classification tree as a hierarchical concurrent state machine. Using this state machine, our new approach then uses a Multi-agent System to generate test sequences by finding and rating valid paths through the state machine.

Keywords: Test sequence generation, classification trees, state machines

1 Introduction

Software has become a central part of our everyday life, both visible to and hidden from human notice. Software controls alarm clocks, coffee and washing machines, the garage door, or the house alarm system. Car engines have a software driven engine control unit. Car radios use software to play music files from a wirelessly connected mobile phone, which itself contains software. Traffic lights and intelligent traffic signs are controlled by software, as well as GPS devices used to guide our ways. There is already software inside the human body as part of heart pacemakers or intelligent prostheses.

Since software is omnipresent nowadays, software quality becomes crucial. One way to ensure software quality is software testing. Other ways include formal reviews and mathematical proofs.

The classification tree method [GG93] can be used for test planning and test design. It allows for a systematic specification of the system under test and its corresponding test cases.

The classification tree method has been implemented by a graphical editor, the classification tree editor [LW00]. The editor has adopted results from the field of combinatorial interaction testing [NL11], which allows to generate certain test suites automatically after the specification of the system under test has been given.

So while the classification tree method and the corresponding editor are of great help for test engineers, there is still a mayor short-coming: Test sequences (a series of test steps) can only be defined manually. There is no concept for dependency rules between single test steps nor automated test sequence generation in the classification tree editor.

In this work we develop an approach for **test sequence generation** with the classification tree method, which is analogue to existing test suite generation introducing **advanced dependency rules** and **new generation rules**. Advanced dependency rules must allow the specifications of constraints between different test steps of a test sequence while new generation rules will specify coverage levels to be reached with the set of all generated test sequences. For implementation, we will use a multi-agent approach.

2 Design

For our new approach, we want to enable test sequence generation from classification trees. Analogously to existing approaches, we identify three kinds of parameters for test sequence generation, the classification tree itself, dependency rules and generation rules. The classification tree holds all parameters and their corresponding values of the system under test. For dependency rules, we extend existing rules to new rules describing constraints between single test steps. Our new rules apply per test sequence. Within each test sequence, dependency rules must not be violated. The generation rules describe desired coverage levels for the resulting set of test sequences. The set of test sequences as a whole must respect the generation rule.

New dependency rules: Existing dependency rules allow the user to specify constraints between parameter values of different parameters within one test case. With the new extended dependency rules, it should be possible to specify constraints between parameter values from one test step to another. The following types of dependency rules shall be supported (with $i, j, k, n, o \in \mathbb{N}; m \in \mathbb{Z}$):

- If class c_i from classification C is selected in test step t_n, then class c_j from classification C must be selected in the succeeding test step t_{n+1}.
- If $C = c_i$ in t_n, then $C = c_j$ in a later t_{n+m}.
- If $C = c_i$ in t_n, then $C = c_j$ in all t_{n+1} to t_{n+m}.
- If $C = c_i$ in t_n, then $C = c_j$ in all t_{n+m} to t_{n+o}.
- Compositions of any (*AND*, *OR*, *NOT*, *NAND*, *NOR*, *XOR*, ...) combination, e.g. if $C = c_i$ *OR* $B = b_k$ in t_n, then $D =$ *NOT* d_j in a later t_{n+m}.

The existing dependency rules are a subset of our new dependency rules for t_n and t_{n+m} with $m = 0$.

Classic dependency rules are valid for manually created test cases, too. We want our new dependency rules to be available for manually created test sequences, as well.

New generation rules: In analogy to conventional test generation, covering all pairs of transitions between classes of the classification tree could be defined as well. Conventional test case generation supports mixed strength generation as well as seeding, so we require them as well.

New generation rules should allow any *t*-wise coverage for both classes and transitions. Note that some of Kuhn's *t*-way sequences [KKL10] can be mapped to our generation rules.

Kuhn's *1*-way sequence coverage corresponds to *1*-wise (or minimal) class coverage here. Each class is supposed to be in the result set at least once. Our approach extends conventional class coverage for test cases to test sequences.

Kuhn's *2*-way sequence coverage corresponds to our *1*-wise (or minimal) transition coverage. All valid transitions (pairs of states) are supposed to be in the result set at least once. In conventional test case generation, there is no coverage criterion for transitions.

Higher n-way (with $n > 2$) sequence coverage is not yet included and will be future work. Instead, we require higher n-wise (with $n > 1$) coverage for both classes and transitions. We have included pairwise class and transition coverage.

The generation rules should take classifications as parameters to specify their focus. The number of parameters should not be restricted. Elements of generation rules should be combinable to allow mixed strength generation. It should be possible to seed in a set of manually created test sequences. The generation algorithm should then analyze this set and take these sequences into account. The generation should, of course, take the dependency rules into account.

Approach: Our approach for test sequence generation is based on an idea proposed by Conrad [Con05], who suggests that the interpretation of classification trees as parallel FSMs together with a set of test sequences allows measuring coverage levels.

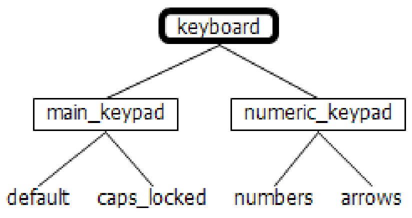

Fig. 1: Classification tree for the keyboard example

Example keyboard states: Given a classification tree *keyboard* (Figure 1) together with a set of (manually specified) test sequences, we can derive a parallel state machine (Figure 2).

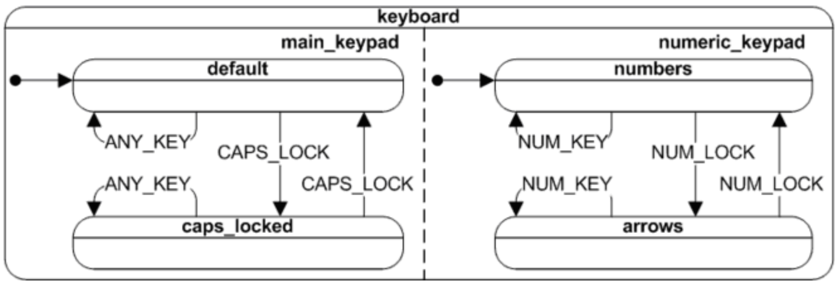

Fig. 2: Parallel FSM for the keyboard example [Mir09]

In UML state charts, parallel states are called orthogonal regions [Obj10, Section 15.3.10].

Conrad's approach, however, lacks some details:

1. He does not give advice how to interpret classification trees with refinements. All trees in his examples are flat trees; there are no refined classes.
2. There is no distinction between directions of transitions. All examples given do not differ between transitions to or from a node. Loops are missing as well.
3. His approach does not handle dependencies. The test engineer has to decide on his own, which combination of classes and which order of consecutive test steps are valid.
4. There is (to our best knowledge) no automatic test sequence generation in Conrad's approach. The test engineer has to specify all test sequences manually.

We will now handle these short comings one-by-one:

1. The interpretation of refined classes can be easily accomplished by mapping them to hierarchical states in state machines. This concept is known from Harel statecharts [Har87], as well. As in classification trees, statecharts can be modeled top-down, from overview to detail, by refining states with a set of substates. This allows different levels of granularity within a single statechart at different hierarchies. We will from now on call the parallel FSM approach the Hierarchical Concurrent finite State Machine (HCSM) approach. Note that we renamed parallel to concurrent, added hierarchies and dropped finite from the abbreviation in analogy to [Luc89].
2. We will differentiate between different transition directions and will enforce loop transitions (transitions, where start-node and end-node are the same) if they exist. For this to work, we will annotate these details to the concerning tree elements.
3. For the handling of dependencies, we will use our own initial dependency rule approach for test sequence generation, here. We will model

transition guards as well using this technique to have only one central dependency handling.
4. The actual test sequence generation will be given in detail next.

For the actual test sequence generation from classification trees, we make some default assumptions:

- In any given plain classification tree, there are no transitions between classes, resulting into an unconnected graph. Test sequence generation will lead to sequences with only one single test step. These test suites are similar to conventional test case generation. Transition coverage is, of course, not available.
- We allow all classes to be reached at start.
- Classifications do not have a (deep) history.

The conversion algorithm from classification trees to HCSM is given next (Figure 3).

The *build*-method takes a tree item as the input parameter and returns its corresponding state machine. First, a list of child states is prepared by recursively building all children of the current tree item.

The *build*-method then distinguishes between several cases: If the current tree item is a class or a composition (Line 9), it distinguishes again between the number of children. If a class or a composition has more than one child (Line 10), then this tree item is a parallel state and all children are partitions of it. If a class or a composition has exactly one child (Line 22), it is skipped by directly adding all the child's children to the current tree item. In all other cases, e.g. the tree item is a class or a composition without any children or the tree item is a classification, the prepared list of children is used as the result's list of children (Line 28).

All transitions and possible start states stored in the classification are read and the result is returned.

3 Implementation

We use two phases and two kinds of agents to traverse the tree. Travelling is done in such a manner that only valid paths are taken and that all travelled paths together already result into the desired coverage, so that there is no need for subset selection.

For the realization, we introduce two kinds of agents: The walker agent and the coverage agent. Both agents will cooperatively traverse the HCSM following the algorithm given in the next section.

Walker agents: The task of this agent is to actually walk through the statechart. The walker agents are very simple. There is only one kind of walker. Walkers do not have a special order; all of them have the same importance. Walkers can have different lifecycles. They are created and removed on demand. There typically is one walker per active state. Walkers

```
build (treeItem)
state = new State
List children = new List()
for all child of treeItem.getChildren do
    childState = build(child)
    children.add(childState)
end for
Boolean addChildren = !children.isEmpty()
if treeItem is Class || treeItem is Composition then
    if treeItem.children.count > 1 then
        addChildren = false
        result = concurrentState(result)
        List subStates = new List()
        for all item of children do
            if item is SubState then
                result.addSubState(item)
            end if
        end for
    else
        if treeItem.children.count == 1 then
            addChildren = false
            result.setChildren(firstChild.getChildren())
            result.setPosition(firstChild.getPosition())
        end if
    end if
end if
if addChildren then
    result.setChildren(children)
end if
readArcs(treeItem, result)
return result
```

Fig. 3: Classification tree to HCSM algorithm

can decide, whether they are stuck, which means, that there is no more (valid) transition available. Walkers have a list of child walker, which can be empty.

Coverage agents: The coverage agents are more sophisticated than the walker agents. Their task is to measure all current and previous coverage and to guide the walkers though the statechart. There are different kinds of coverage agents, one for each type generation-rule term introduced in Section 2: State-coverage, transition-coverage, state-pair-coverage, transition-pair-coverage and so on. We define an order over all coverage agents by a) their complexity and b) the number of parameters (which is the number of scopes they cover). The lifecycle of the coverage agents start with the beginning of the generation process. They remain active until the coverage criterion they handle is finished. There is one coverage agent per generation rule term/component, e.g. the rule

$$state - pairwise(a, b) + transitions(a, b, c)$$

results into two coverage agents.

Coverage agents can decide, whether they are finished (both globally and locally). From this example above, we can explain complexity and number of parameters, needed for ordering the coverage agents. We use the following formula to calculate the order of coverage agents:

$$i = |order| * |parameters|$$

If two terms have the same index value, we will prefer those term being early in the generation rule formula. This can result into missing commutative property for certain generation rules.

Walker algorithm: The general idea of the algorithm (Figure 4) is that all walkers will, one by one, ask the most complex remaining coverage agent where to go next. By walking the route proposed by the most complex coverage agent first, chances are high to cover elements needed by simpler coverage agents, too. For example, transition coverage for a statechart already implies state coverage, too.

Very first step is the creation of coverage agents from the generation rule. Then, the root node (*statechart*) is selected. This means, a walker is created for the root node. For each classification (*parallel section*) under the root node, the class (*node*) with the start transition is selected (Lines 4-9). There will be one walker per classification now. The selection of start nodes is repeated while a selected class (*state*) has at least one refinement (*inner states*). A new walker is created for each refinement step. It will be added to the list of child walkers.

When all walkers without child-walkers are on an atomic class, the first test step is created. This test step is then checked against the conventional dependency rules. If it is valid, then the test step is added to the test sequence (Line 12) and the coverage measure is updated (Line 13). If it is invalid, then the test sequence generation is cancelled and an empty result set is given (Line 15) as there are no valid test sequences available from this specification.

While generation is not finished, each walker without child walkers will now perform the following steps. It first identifies the most complex remaining coverage agent (Line 20). It will then ask this one agent where to go next (or to stay) (Line 21). The candidate class is then checked against conventional and new dependency rules together with all walkers already moved in this turn. If the candidate is valid, we continue with the next walker until all walkers have walked. We then add this test steps to the test sequence (Line 32) and update the coverage. We then start the next turn of walking walkers. If the candidate node does not offer any valid test step, the walker will take the second best move from the coverage agent candidates and validates it again until a valid test step candidate is found. If all candidates

```
 1: create coverage agents from rule
 2: select root node // statechart
 3: create walker
 4: for all classification // parallel section do
 5:    repeat
 6:       select (inner) start class // node
 7:       create walker, add to walker child list
 8:    until class is atomic
 9: end for
10: compose test step from selected classes
11: if valid then
12:    add test step to sequence
13:    update coverage
14: else
15:    cancel generation with empty results
16:    finished = true
17: end if
18: while not finished do
19:    for all walker without child-walker do
20:       find coverage agent
21:       ask coverage agent where to go or stay
22:       decide if stuck
23:       if walk transition then
24:          while entering a refined class do
25:             select (inner) start class
26:             create walker, add to walker list
27:          end while
28:       end if
29:    end for
30:    compose test step from selected classes
31:    if valid then
32:       add step to sequence
33:       update coverage
34:    else
35:       backtracking
36:    end if
37: end while
```

Fig. 4: Test sequence generation algorithm, phase 1

fail, the walker will try to stay where it is. If that is not possible, it can still try to step out, which means, its parent walker will move next. If even that is not an available option, we start backtracking (Line 35).

The candidate from the previous walker is rejected now and the next best choice from this previous walker is taken. This is done recursively until one valid candidate can be found. If there is no candidate (the first walker does not find a valid option), then the statechart is globally stuck. It will be reset to its initial state by turning it off and on again.

If adding test steps to the test sequence does not increase coverage for a certain number of steps, we will reset the state chart as well and start a new test sequence. We wait as many steps as there are (inner) classes in the largest classification. All steps without progress are removed from the sequence before adding it to the result test suite.

We repeat this until the global coverage is completed, then we can stop, or if resetting and starting a new sequence does not increase coverage for a certain number of test sequences. We again wait as many test sequences as there are (inner) classes in the largest classification. In this case, we start a second phase for hard to reach configurations.

```
 1: for all coverage agent (in descending order of importance) do
 2:     for all uncovered item do
 3:         find path from item to start in a reverse breadth first search
 4:         if there is valid path then
 5:             add test sequence of path to result set
 6:         else
 7:             drop item from coverage measure
 8:         end if
 9:     end for
10: end for
```

Fig. 5: Test sequence generation algorithm, phase 2

Second phase: The algorithm for the second phase for hard to reach configuration is given in Figure 5.

The approach is rather simple here. For all coverage agents we get all uncovered items. We do a reverse breadth first search for paths from this item to possible start states (Line 3). If a valid path is found, it is added to the result set (Line 5). If there is no valid path, the item is not reachable and is dropped (Line 7).

The approach does guarantee the coverage of all coverable items. The result set however might not be minimal.

Coverage algorithm: The coverage rate algorithm works as follows (Figure 6). It gets a candidate state or transition. For state coverage, self transitions are ignored and zero is returned. Otherwise, it then adds this candidate to a queue together with a weight factor, with an initial weight factor of one. The initial rating is set to zero. The candidate is added to the list of rated items. Then while the queue is not empty the algorithm polls the next state and weight factor from the queue. If the polled node is the original candidate and if the rating is larger than zero, the algorithm has found a loop path with new items. This loop path is prefered by adding the value of 100 to the rating. In this case or when the current item is on the list of rated items, the while loop steps to its next cycle. Else this node is added to the list of rated items. If the node is on the list of target states (it has not

```
candidate state or transition
if state coverage && self transition then
    return 0
end if
weight = 1.0
rating = 0
queue += (candidate, weight)
while !queue.empty do
    (item, weight) = queue.poll
    if item == candidate && rating > 0 then
        rating += 100
        continue
    end if
    if ratedItems contains then
        continue
    end if
    ratedItems += item
    if targetNodes contains item then
        rating += 10 * weight
    end if
    if item has (outgoing transition || childnodes || subsections) then
        weight *= 0.95
    end if
    for all (outgoing transition && childnodes && subsections) of item do
        queue += (item, weight)
    end for
end while
return rating
```

Fig. 6: Rating of candidates

been used in any test step before), the algorithm adds 10 times the weight factor to the result rating. Then if there are outgoing transitions, child nodes or subsections, the weight factor is multiplied with a punishment value of 0.95. Target states of outgoing transitions, child nodes and subsections are then added to the queue together with the new weight factor. When the queue is empty the rating is returned.

4 Evaluation

We use the Keyboard example [Mir09] (as given in Figure 2). We found some more examples in Literature, a Microwave [Luc89], an Autoradio [Hel07], and of course, Harel's Citizen watch [Har87].

From the IBM Rhapsody examples, we took the Coffee Machine, the Communication example, the Elevator, and the Tetris game. In Matlab Simulink Stateflow, we found Mealy Moore, Fuel Control, Transmission, and Aircraft.

Details on case studies and results are given in Table 1.

Table 1: Results for test sequence generation

Name	States	Transitions (without start)	Minimal	Pairwise	Complete	State Coverage	Transition Coverage
Keyboard [Mir09]	5	8	2	4	4	2	5
Microwave [Luc89]	19	23	7	28	56	9	17
Autoradio [Hel07]	20	35	11	33	66	13	36
Citizen [Har87]	62	74	31	108	3121	47	51 *(92.7%)*
Coffee Machine	21	28	9	27	81	9	18
Communication	10	12	7	NA	7	7	17
Elevator	13	18	5	20	80	6	9
Tetris	11	18	10	NA	10	15	31
Mealy Moore	5	11	5	NA	5	5	24
Fuel Control	5	27	5	25	600	5	12
Transmission	7	12	4	12	12	4	9
Aircraft	24	20	5	31	625	4 *(86.2%)*	7 (2)

The set of case studies is still too small to make final statements on performance, regarding scalability, execution times and result set sizes. Preliminary results are, however, already very promising.

Execution, times: We are not giving detailed figures for generation times here since they are all below 1sec on an Intel Core2Duo with 2Ghz and 3GB RAM running our java implementation on a single core under Windows XP.

Evaluation, coverage: Results from the experiments clearly show that our approach is capable of generating test sequences with given coverage levels. For State coverage, 100% coverage was reached for 11 of 12 scenarios. The remaining scenario only resulted into 86.2% coverage (scenario Aircraft). In all 12 cases, coverage was reached in a single test sequence.

For Transition coverage, again for 11 of 12 cases 100% coverage has been reached. For the Citizen scenario, only 92.7% coverage was archieved. In 11 of 12 cases, the result only consists of one test sequence, while for the aircraft scenario two sequences were generated. The algorithm has reseted the walker agents to their initial positions to reach missing states. Note: Results given here only contain implementation of walker phase one, since phase two was not available yet.

For 22 of 24 cases, full coverage was reached. We are certain to complete the two remaining scenarios with phase two implementation.

Evaluation, size: Test result sizes are not evaluated in our case studies since there is no comparison available to other approaches yet. For the goal of test suite minimization we will need to evaluate minimal result suite size using a brute force approach, allowing us to compare our results For the brute force approach, we expect to see long execution times making it impractical for product use.

Scalability: We have not yet tested our approach for very large scale case studies. If performance decrease is too large, we might reduce the search depth of coverage agents.

Extendability: We have successfully implemented test sequence generation for state and transition coverage. Next steps will be the generation of state pair coverage and transition pair coverage. The implementation will be more difficult duo to the nature of pairwise coverage problems being known NP-complete [WP01].

Parameter tuning: We have done experiments on the influence of the punishment factor for both State and Transition coverage. In a set of 100 experiments each, we have tested all factors from 0.01 to 1.00 for all 12 scenarios. Lower values like 0.1 turns out to be much better than 0.95 as it was selected earlier. However for one scenario, the Communication example. higher punishment values result into smaller test suites and better coverage. We need to investigate the influence of the other parameters as well.

5 Related Work

In [CDFY99], Conrad et al. present the automatic import of Simulink Models into classification trees. They use imported models for systematic determination of test scenarios. Test scenarios can be either test sequences or test cases.

Test scenarios are series of stimuli in a certain order and with durations assigned. They use the combination table of the classification tree editor to define signal courses. With additional metadata, they annotate the type of the course. Supported course types are step, ramp and spline, with step being the default. Their system can be used to model both discrete as well as continuous signals. The reuse of modeling information from the development for test activities reduces time and costs for test modeling.

The approach is further described in [Con05]. The work focuses on the integration of test scenarios for embedded systems into the development process. Model-based specification, design and implementation are in place, but testing still can be improved. Since testing all feasible combinations is nearly impossible, a good selection of test scenarios determins extent and quality of the whole test. The automatic creation is desirable but only possible to a limited extent yet. Which leads to largely manual test design. The problems of ad-hoc test scenario selections are redundancy and possible gabs. They are typically in a very concrete notation with a low level of abstraction, making reuse difficult.

The proposed solution is a model based testing (MBT) approach [EFW01] based on an abstract model of the input data. The input partitioning of this approach implies a parallel state-machine model. Each classification forms one of the parallel parts (AND-states) of the state-machine. The states denote the individual equivalence classes defined for the classification. Test sequences can be viewed as paths through such a test model.

Making the underlaying test model explicit allows comparing Conrad's approach with other MBT approaches. Furthermore, the test model can be used to formalize different coverage criteria. Conrad suggests the systematic scenario selection based on the functional specification.

Current tools supporting the classification tree method do only allow manual definition of test sequences. There are no generation rules for test sequence generation; desired coverage levels for a set of test sequences cannot be specified. Dependency rules to describe constraints between single steps of test sequences do not exist and can therefore not be checked.

Heimdahl et al. briefly surveys a number of approaches in which test sequences are generated using model checking techniques [HRV+03]. The common idea is to use the counter-example generation feature of model checkers to produce relevant test sequences.

Model checking aims to prove certain properties of program execution by completely analyzing its finite state model algorithmically [BE10,JM09]. Provided that the mathematically defined properties apply to all possible states of the model, then it is proven that the model satisfies the properties. However, when a property is violated somewhere, the model checker tries to provide a *counter-example*. Being the sequence of states the counter-example leads to the situation which violates the property. A big problem with model checking is the *state explosion problem*: The number of states may grow very quickly when the program becomes more complex, increasing the total number of possible interactions and values. Therefore, an important part of research on model checking is *state space reduction*, to minimize the time required to traverse the entire state space.

The *Partial-Order Reduction* (POR) method is regarded as a successful method for reducing this state space [JM09]. Other methods in use are *symbolic model checking*, where construction of a very large state space is avoided by use of equivalent formulas in propositional logic, and *bounded model checking*, where construction of the state space is limited to a fixed number of steps.

A technique for test sequence generation is introduced by Kuhn et al.: They generate event sequences for a given set of system events. They allow specifying t-way sequences, which includes all t-events being tested in every possible t-way order [KKL10].

Several researchers propose other approaches for test sequence generation. Wimmel et al. [WLPS00] propose a method of generating test sequences using propositional logic.

Ural [Ura92] describes four formal methods for generating test sequences based on a finite-state machine (FSM) description. The question to be answered by these test sequences is whether or not a given system implementation conforms to the FSM model of this system. Test sequences consisting of inputs and their expected outputs are derived from the FSM model of the system, after which the inputs can be fed to the real system implementation. Finally, the outputs of the model and the implementation are compared.

6 Conclusion

We have successfully implemented test sequence generation for the classification tree method using a multi-agent system. The distinction between simple walker agents and sophisticated coverage agents enables us to generate test suites with desired coverage levels. Preliminary results are already very promising. The dependency rules and generation rules turn out to work pretty well. They offer a decent granularity to describe all desired scenarios. Since we split actual traversal of the test problem from the rating of possible paths, we can now easily swap in single parts, e.g. use different guidance in the coverage agents.

For future work, we see the extension to higher n-wise coverage, completition of incomplete scenarios and a large scale set of benchmarks. As already proposed in [KL11], we will use search based techniques as well to evaluate effectiveness and efficiency of our approach. Next steps are the limitation of sequence lengths and favoring of reset or travel back to start, which is considering the cost of a reset. We will also evaluate the influence of parameter tuning for result set sizes.

Acknowledgments. Work supported by EU grant ICT-257574 (FITTEST).

References

BE10. Dragan Bošnački and Stefan Edelkamp. Model checking software: on some new waves and some evergreens. *Int. J. Softw. Tools Technol. Transf.*, 12:89–95, May 2010.

CDFY99. Mirko Conrad, Heiko Dörr, Ines Fey, and Andy Yap. Model-based Generation and Structured Representation of Test Scenarios. In *Proceedings of the Workshop on Software-Embedded Systems Testing, Gaithersburg, Maryland, USA*, 1999.

Con05. Mirko Conrad. Systematic testing of embedded automotive software-the classification-tree method for embedded systems (CTM/ES). *Perspectives of Model-Based Testing*, 2005.

EFW01. Ibrahim K. El-Far and James A. Whittaker. Model-Based Software Testing. *Encyclopedia of Software Engineering*, 2001.

GG93. Matthias Grochtmann and Klaus Grimm. Classification trees for partition testing. *Softw. Test., Verif. Reliab.*, 3(2):63–82, 1993.

Har87. David Harel. Statecharts: a visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.

Hel07. Steffen Helke. *Verifikation von Statecharts durch struktur- und eigenschaftserhaltende Datenabstraktion*. PhD thesis, Technische Universität Berlin, 2007.

HRV+03. Mats P.E. Heimdahl, S. Rayadurgam, Willem Visser, George Devaraj, and Jimin Gao. Auto-generating test sequences using model checkers: A case study. In *3rd International Worshop on Formal Approaches to Testing of Software (FATES 2003)*, 2003.

JM09. Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41:21:1–21:54, October 2009.

KKL10. D. Richard Kuhn, Raghu N. Kacker, and Yu Lei. Practical combinatorial testing. Technical report, National Institute for Standards and Technology (NIST), October 2010.

KL11. Peter. M. Kruse and Kiran Lakhotia. Multi objective algorithms for automated generation of combinatorial test cases with the classification tree method. In *Symposium On Search Based Software Engineering (SSBSE 2011)*, 2011.

Luc89. Paul J. Lucas. *An Object-Oriented Language System For Implementing Concurrent, Hierarchical, Finite State Machines*. PhD thesis, Graduate College of the University of Illinois at Urbana-Champaign, 1989.

LW00. Eckard Lehmann and Joachim Wegener. Test case design by means of the CTE XL. In *Proceedings of the 8th European International Conference on Software Testing, Analysis & Review (EuroSTAR 2000), Kopenhagen, Denmark*. Citeseer, 2000.

Mir09. Mirosamek. Two orthogonal regions (main keypad and numeric keypad) of a computer keyboard. http://en.wikipedia.org/wiki/File:UML_state_machine_Fig4.png, 2009.

NL11. Changhai Nie and Hareton Leung. A survey of combinatorial testing. *ACM Comput. Surv.*, 43:11:1–11:29, February 2011.

Obj10. Object Management Group. *OMG Unified Modeling Language (OMG UML), Superstructure Verion 2.3*, 2010.

Ura92. Hasan Ural. Formal methods for test sequence generation. *Comput. Commun.*, 15:311–325, June 1992.

WLPS00. Guido Wimmel, Heiko Loetzbeyer, Alexander Pretschner, and Oscar Slotosch. Specification based test sequence generation with propositional logic, 2000.

WP01. Alan W. Williams and Robert L. Probert. A measure for component interaction test coverage. In *Proc. ACS/IEEE Intl. Conf. on Computer Systems and Applications*, volume 30, pages 301–311, 2001.