Minimizing Test Execution Time During Test Generation

Tilo Mücke and Michaela Huhn

Technical University of Braunschweig, 38106 Braunschweig, Germany
{tmuecke,huhn}@ips.cs.tu-bs.de,
WWW home page: http://www.cs.tu-bs.de/ips

Abstract. In the area of model based testing, major improvements have been made in the generation of conformance tests using a model checker. Unfortunately, the execution of the generated test suites tend to be rather time-consuming. In [1] we presented a method to generate the test suites with the shortest execution time providing the required coverage, but this method can only be applied to small models due to memory-consumption. Here we show how to generate test suites for a number of different test quality criteria like coverage criteria, UIOs, mutant testing. Moreover, we present heuristics to significantly reduce test execution time that are as efficient as a naive testsuite generation. Our optimization combines min-set-cover-algorithms and search strategies, which we use to enlengthen generated test cases by promising additional coverages. We compare several heuristics and present a case study where we could achieve a reduction of the test execution time to less than 10%.

1 Introduction

In the last decade, models have been discovered as an invaluable source for deriving test cases. Many authors proposed model checking [2–4] or other search strategies [5] to automatically generate test sequences from behavioral (semi-)formal models. The success of model based testing has its reasons in the wide acceptance of model based development in practice, in particular in the embedded domain where substantial verification and testing of systems is obligatory.

We present an approach to model based test generation that uniformly handles a number of well-established test quality criteria like test purposes [5], coverage criteria [4], and mutation testing [6] and applies them on behavioural models. Our key technology for test case generation is model checking on state based systems. In a preparatory step, test quality criteria are split into subgoals that can be achieved by a test case and the models are instrumented by adding auxiliary variables and test drivers to direct the search for test cases to the subgoals. Using this procedure systematically, large testsuites for involved test quality criteria can be generated automatically.

However, many generated testsuites expose long test execution times which is a limiting factor in several real-time domains: For instance, in railway interlockings, traffic or process control systems some actions need relevant time for execution. We address this problem by combining heuristic search algorithms with min-set-cover algorithms for minimizing the test *execution* time of the testsuites but preserving the test quality.

A second problem of automated test generation is the fault recognition rate. As recently observed by Heimdahl [7], structural test quality criteria tend to produce testsuites that just satisfy the criteria instead of verifying the correct behavior. We cope with this weakness by using strong quality criteria [8], by enlengthening the test cases with UIOs (see section 2.6), and by extending test cases gathering additional coverage (see section 3). Consequently, our approach has the potential to generate test cases with a higher fault recognition rate.

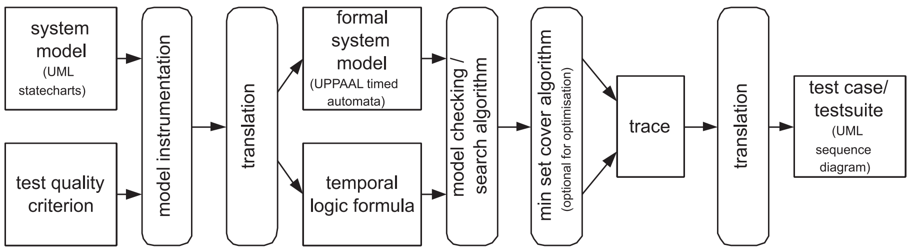
system model (UML statecharts)
test quality criterion
model instrumentation
translation
formal system model (UPPAAL timed automata)
temporal logic formula
model checking / search algorithm
min set cover algorithm (optional for optimisation)
trace
translation
test case/ testsuite (UML sequence diagram)

Fig. 1. Automated model based test case generation

Figure 1 summarizes the procedure for test case generation. Initially, the instrumentation of the model has to be adapted according to the selected test quality criterion. The different instrumentations are described in Section 2. The details on the translation from UML statecharts [9] to the input language of the UPPAAL model checker [10] and back can be found in [1]. Section 3 is concerned with the combination of search heuristics for test case generation and min-set-cover algorithms to optimize testsuites with respect to test execution time. The results of a case study are reported in Section 4 and we conclude in Section 5.

2 Test Case Generation

In this section we describe the instrumentation of models for test case generation via model checking for three different test quality criteria:

1. A *test purpose* is given by a test expert. It consists of a desired property or a critical operation sequence. Test cases checking the property or executing the sequence are generated.
2. *Coverage criteria* are definitions of model element type dependent, structural properties which have to take place during test case execution. E.g. state coverage demands each state to be visited at least once.
3. *Mutant testing* demands that every mutated model has to be uncovered as erroneous by the testsuite, unless it behaves equivalent to the original model. The mutants are generated automatically by so called mutation operators. E.g. arithmetic operator replacement (AOR) replaces each occurrence of an arithmetic operator by any other arithmetic operator.

We decompose each test quality criterion into subgoals which are to be achieved and call them *partial coverages*. Partial coverages are encoded as predicates that shall be satisfied on some execution like "*state s has to be reached*". For our purposes, the system model consists of a family of UML statecharts [9] modeling the behavior of the system components. UML statecharts extend final state machines by the concepts of hierarchy, concurrency and communication via events. Transitions can be labeled with events, guards and actions. Additionally, we use a time event after(t) with the obvious meaning. There exist a number of formal semantics for statechart dialects and we will use the approach from [11, 12] to transform statecharts into the input language of the UPPAAL model checker [10] and translate the output traces of the model checker back into sequence diagrams. UPPAAL supports the verication of real-time constraints, a feature we use for the generation of time annotated test cases. The real-time annotations within the models result from measuring and approximating the execution times of actions from previous versions of the components. Test purposes are given in terms of predicates or UML sequence diagrams for scenarios. To illustrate test case generation for the quality criteria, we use a simple running example: The control of a dimmer switch.

2.1 Example: Dimmer Switch

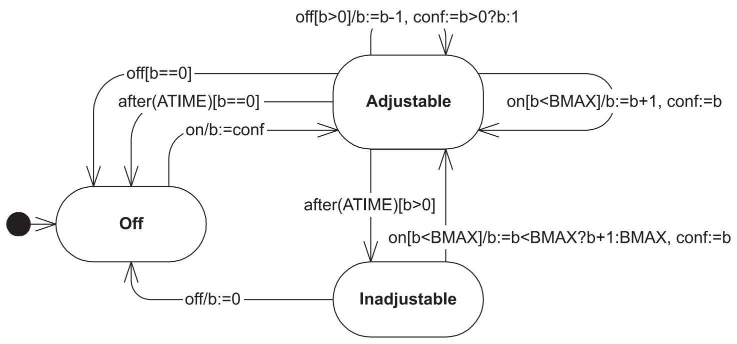
off[b>0]/b:=b-1, conf:=b>0?b:1
off[b==0]
after(ATIME)[b==0]
on/b:=conf
Adjustable
on[b<BMAX]/b:=b+1, conf:=b
Off
after(ATIME)[b>0]
on[b<BMAX]/b:=b<BMAX?b+1:BMAX, conf:=b
off/b:=0
Inadjustable

Fig. 2. Statechart of the dimmer switch

The dimmer switch is controlled by two buttons (events): on and off. The brightness b of the connected lamp is the only observable output of the system and can be adjusted within the values 0 (light off) and BMAX. When the on button is pushed, the lamp lights with the brightness b that is memorized (conf) from the previous use. After turning the dimmer on, the brightness can be modified smoothly by pressing the buttons on and off. If for the time ATIME no button is pressed, the brightness will become inadjustable. Pushing the off button turns the lamp off immediately. By pressing the on button, the lamp returns into adjustable mode. The statechart model of the dimmer switch is shown in Fig. 2.

2.2 Test Driver

For test case generation, a test driver has to be added which feeds the system under test with all possible inputs. This test driver has to be implemented non deterministically (see figure 3). Technically, the test driver is put in parallel to the statecharts modeling the system which leads to a product construction at the level of model checking. Fortunately, the test driver consists of only one state.

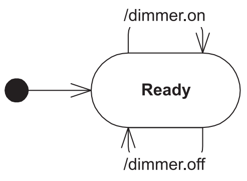

Fig. 3: Test driver

2.3 Test Purpose: Property

First we consider properties that shall be checked by tests. An example property for this model is:

	temporal logic formula	meaning
1	$E <> (b > 0)$	The lamp can be turned on.

In case a property can be directly expressed as a path quantified state predicate the model instrumentation can be omitted. The only thing to do is to add a query at the level of the model checker, i.e., $E <> p$ (on some trace p happens) in UPPAAL syntax. For a property $E <> p$ the model checker returns a trace if it is satisfiable.

2.4 Test Purpose: Interaction Sequence

Often a test purpose can be naturally described as a sequence of operations or interactions. Thus, sequence diagrams are a widely used representation in testing which we use as well. Figure 4 shows a sequence diagram where on is pressed, it is waited ATIME and then off is pressed, demanding that the brightness is set to 0. Afterwards conf is set to 2 and on is pressed, so that b will be 2.

To generate a test case including this sequence, the test driver is modified. It consists now of it's nondeterministic part which is needed to reach a state where the sequence can be executed and a simple representation of the activities in the sequence diagram from the point of view of the test driver, see Figure 4 (right). To generate a test case containing this sequence we ask the model checker for a path reaching the partial coverage: $E <> testdriver.Finished$.

Possible actions of the test driver are (1) sending and (2) receiving events, (3) changing and (4) monitoring global variables, and (5) evaluating time constraints. How these actions are translated from sequence diagrams in test drivers is formally described in [11]. Time constraints are realized during translation to UPPAAL timed automata.

To generate more than one trace per sequence diagram, one can vary the state at which the execution of the sequence starts which will lead to additional partial coverages representing different settings where the scenario is executed.

2.5 Coverage Criteria

A model checker can as well be used to generate test cases for coverage criteria. In [1] it is shown, how models are instrumented and properties are generated to achieve state coverage, transition coverage, modified condition/decision coverage, boundary coverage and dataflow coverage. The instrumentation adds new coverage variables to the system, which are set, whenever a partial coverage is achieved. E.g. to achieve transition coverage each transition is once supplemented with a statement setting the coverage variable to true. Thus with a query $E <> (coverageVar)$ the appropriate partial coverage can be achieved. Other coverage criteria require splitting of transitions (modified condition/decision coverage) or adding auxiliary variables (dataflow coverage). The new coverage criterion boundary coverage is now used as an example to show an instrumented version of Fig. 2.

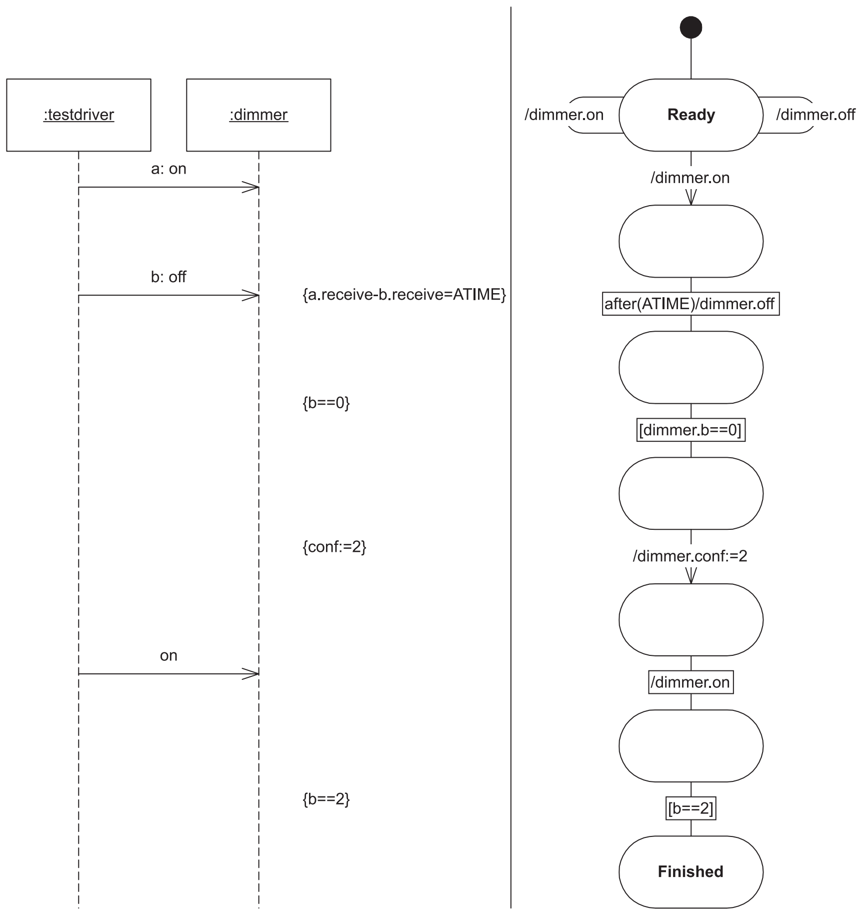

Fig. 4. Statechart of a modified test driver for the sequence diagram

Boundary Coverage demands that for each guard containing a relational operator a test case is generated for which the operands are as close as possible to the boundary. To achieve this criterion, the structure of the statechart is changed by splitting transitions. Each transition with a relational operator in its guard is split in two transitions, one of which fires for the closest operands possible only and is instrumented by a coverage variable and another one which conserves the behavior by firing in all other possibilities. Figure 5 shows, how the Dimmer Switch is instrumented to enable the model checker to generate test cases using the testdriver from Fig. 3 and the queries:
$E <> B1 == true$, $E <> B2 == true$, and $E <> B3 == true$.

2.6 Unique Input Output Sequences

Recent results by Heimdahl [7] indicate that a testsuite generated for structural coverage criteria like state or transition coverage may be weak w.r.t. its fault detection ability. But fault detection is the overall aim of testing.

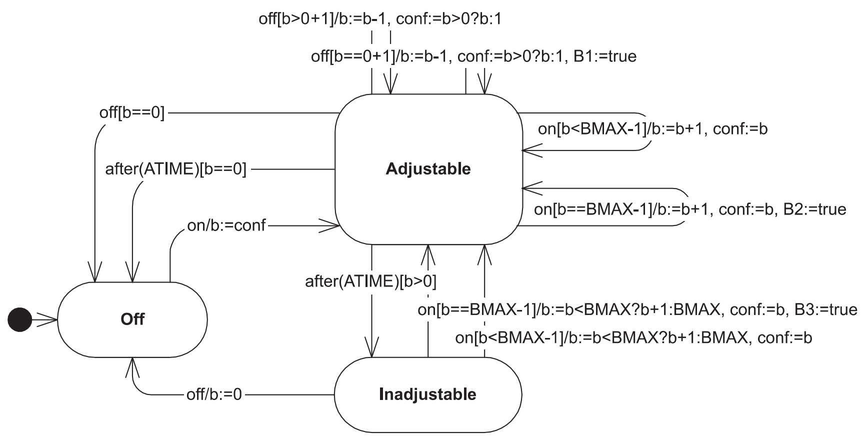
off[b>0+1]/b:=b-1, conf:=b>0?b:1
off[b==0+1]/b:=b-1, conf:=b>0?b:1, B1:=true
off[b==0]
after(ATIME)[b==0]
on/b:=conf
Adjustable
on[b<BMAX-1]/b:=b+1, conf:=b
on[b==BMAX-1]/b:=b+1, conf:=b, B2:=true
after(ATIME)[b>0]
on[b==BMAX-1]/b:=b<BMAX?b+1:BMAX, conf:=b, B3:=true
on[b<BMAX-1]/b:=b<BMAX?b+1:BMAX, conf:=b
Off
off/b:=0
Inadjustable

Fig. 5. Statechart of the Dimmer Switch instrumented for boundary coverage

Let us consider a generation procedure for a testsuite satisfying transition coverage. It will generate test cases in which the last step is the execution of the desired transition. Thus, it is not tested if the correct target state of the transition is actually reached. To overcome this kind of problems, we enlengthen the test cases by so called UIOs (Unique Input Output sequence). These UIOs consist of a sequence which can only be executed starting at the specific state. From all other states, provided they are not trace equivalent, outputs of the system will differ from the outputs described in the UIO.

We generate UIOs using model checking in a variant of [13] by putting several systems with partly modified initial states in parallel. The test driver is modified, so that all systems are triggered with the same inputs. A comparator compares the outputs of the systems. As soon as a system has produced different output compared to all other systems, a UIO for the initial state of this system is found.

2.7 Mutant Testing

Mutant testing requires, that each mutation of the original model is either equivalent to the original model or found as erroneous by the test suite. The mutants are generated by mutation operators.

The classical mutation operators [6] can be applied to code only. Thus, in the case of statecharts, they can be applied to guards and actions. Common mutation operators are: (1) LCR, AOR, ROR which replace each occurrence of a logical, arithmetical, or relational operator by any other operator of the same type, (2) UIO which negates, increments and decrements each arithmetic expression, (3) AAR, ACR, ASR, ... replace each occurrence of a variable, constant or array by each compatible variable, constant or array, (4) CRP, DSA slightly change constant values, (5) SDL deletes each statement, and many more.

On the model-layer, these mutation operators have to be supplemented by mutation operators working with the structure of the automata [14]. Some of these operators are already handled by mutating guards and actions. Examples for other operators are: (1) state missing, (2) transition missing, (3) replace origin state of a transition, (4) replace target state of a transition, (5) replace triggering event, (6) replace triggered event, and (7) replace event recipient.

To generate mutant killing test cases, the original model has to be executed versus a mutated program. They both have to be triggered by the same inputs. A mutant is found as erroneous, if the outputs differ. Thus, the same technique we used for UIO generation is applied for generating mutant killing test cases.

Detecting unsatisfiable coverages is done by using the query $A[]c == \mathit{false}$ for a partial coverage c. If the subgoal cannot be achieved, it is is eliminated.

3 Strategies to Generate Time Optimized Testsuites

After instrumenting the model according to a test quality criterion, the statecharts modeling the system are transformed into a semantically equivalent family of timed automata that serves as formal system model for UPPAAL. Details on the construction, e.g. syntactic restrictions on the UML model elements, the translation of timing constructs, the handling of event queues and UML run-to-completion semantics, can be found in [11].

In [1] we investigate a technique to generate the *time optimal* testsuite. We achieved a significant reduction in test execution time, but the technique suffers from high memory consumption, i.e. it is restricted to small models.

Alternatively, we consider heuristics to efficiently generate testsuites with optimized but not necessarily minimal execution time. We generate an optimized testsuite in two steps:

1. We generate a test case for each partial coverage we are interested in, thereby following the work of Hong et.al. [4]. These test cases build a testsuite for a given test quality criterion as each required partial coverage is achieved at least once. Moreover, some partial coverages may be satisfied by more than one test case[1]. To increase the basis for optimization we enlengthen the test cases by adding a path starting from its final state and leading to a state where some additional coverage is satisfied. We consider several search heuristics to generate a testsuite of promising long test cases. The amount of redundancy with respect to the achieved coverages is controlled by a parameter of the search.
2. We use min-set-cover-algorithms [15, 16] to optimize test execution time of a testsuite but retain the required partial coverages. Since the testsuite has been enlarged, a min-set-cover-algorithm works on a broader basis from which it eliminates test cases that are redundant w.r.t. the achieved coverages. Again we consider different heuristics to improve test execution time.

3.1 Building a Redundant Testsuite

For a succinct description of the heuristic search we use pseudo code. The basic function is called *tcGenerate* and generates a *single test case* as follows: The model is instrumented and transformed to the model checker input language, then the model

[1] A test case for the coverage c_1 may reach other coverages on the way.

checker is employed for searching a trace and finally the resulting trace is retranslated into a sequence that can be executed on the system model. The concepts needed to realize *tcGenerate* have been described in Section 2:

$$tcGenerate : Models \times (Traces \cup \{\varepsilon\}) \times PC \to Traces \cup \{\varepsilon\}$$

tcGenerate takes a statechart model $m \in Models$, an initial segment of a test case $t \in Traces \cup \{\varepsilon\}$[2] and a partial coverage pc from the set of interesting partial coverages PC. It returns a test case tc that extends t, i.e. $tc = t \cdot u$ for some suffix u, and satisfies the given partial pc, if possible. Otherwise, it returns ε. With each generated test case tc we store two attributes: $t_{exec}(tc)$, the test case execution time, and $subPC(tc)$, the subset of PC that is achieved when executing tc.

Now we consider three search strategies that use *tcGenerate* to generate *testsuites* for a given model and a set of partial coverages PC:

$$search : Model \times \wp(PartCov) \to \wp(Traces)$$

The naive search strategy simply generates one test case for each required partial coverage $pc \in PC$ by successively calling *tcGenerate*. Thus the size of the testsuite is $O(|PC|)$.

The depth 2 search aims to enlengthen a test case by a suffix that achieves an additional partial coverage. The initial parts are generated by the naive search strategy and then an extension for each partial coverage is searched:

```
function depth2Search(model, PC)
  testsuite=naiveSearch(model, PC); extension=$\emptyset$;
  foreach testcase $\in$ testsuite do
    foreach pc $\in$ PC do
      extendedTestcase=tcGenerate(model, testcase, pc);
      if (extendedTestcase$\neq\varepsilon$) then extension=extension $\cup$ {extendedTestcase};
    od;
  od;
  return testsuite $\cup$ extension;
```

The number of generated test cases is in $O(|PC|^2)$.

Heuristic Search enlengthens only the best test cases for a partial coverage which has been achieved rarely so far. Therefore we need two ranking functions. $getBestTestcase$[3] returns the test case with the maximal value for $|PC(tc)|/(t_{exec}(tc) + t_{reset})$ which has not been enlengthened in all possible ways. The function $getWorstPartialCoverage$ gives us the partial coverage which has been achieved least often in the testsuite and in particular not in the chosen test case. The parameter *amount* controls how many test cases are generated. In our experiments we used $10 \cdot |PC|$ which seems to generate a sufficiently large set for the subsequent reduction phase.

[2] If t is given as sequence diagram the model is instrumented as described in Sec. 2.4.

[3] A promising test case achieves many partial coverages in short execution time.

```
function HeuristicSearch(model, $PC$)
  testsuite=naiveSearch(model, $PC$);
  for i=1 to amount do
      testcase=getBestTestcase(testsuite);
      pc=getWorstPartialCoverage(testsuite);
      extendedTestcase=tcGenerate(model, testcase, pc);
      if (extendedTestcase$\neq \varepsilon$) then testsuite=testsuite $\cup$ {extendedTestcase};
  od;
  return testsuite;
```

3.2 Optimizing Testsuites by Min-Set-Cover-Algorithms

Originally, a min(imal)-set-cover algorithm constructs a small subset from a set of sets, such that the union of sets in the small subset equals the union of the sets in the original set. Since the minimal set cover problem is NP-complete [17], heuristic algorithms are used. Here we want to adopt min-set-cover algorithms to eliminate test cases, that are redundant w.r.t. the partial coverages they achieve, from the testsuite, thereby reducing test execution time of the testsuite.

$$minsetcover : \wp(Traces) \times \mathbb{R} \rightarrow \wp(Traces)$$

minsetcover takes a testsuite and the reset time t_{reset} of the system that has to be added after each test case to sum up the execution time of the testsuite.

A simple Greedy algorithm can be applied on a minimal set cover problem by complementing the usual approach [15], i.e., we start with the full testsuite and try to eliminate test cases but keep the same coverage. To select the next candidate for elimination the function $entry$ is used. In the simplest variant of a greedy algorithm we set $entry(testsuite, \ldots, i, \ldots) = testsuite_i$.

```
function greedyMinSetCover(testsuite, $t_{reset}$)
  reducedTestsuite=testsuite;
  for i=1 to |testsuite| do
      testcase=entry(testsuite, reducedTestsuite, i, $t_{reset}$);
      if (|testsuite.subPC|==|(reducedTestsuite\{testcase}).subPC|)
          then reducedTestsuite=reducedTestsuite\{testcase};
  od;
```

The bidirectional Greedy Algorithm (see [15]) uses the same $entry$ function but is applied to the testsuite twice running from both directions through the testsuite.

A sorted Greedy algorithm improves testsuite optimization even further by using a better *entry* function that sorts the test cases according to their quality, weakest test case are returned first. For this purpose, the function $getBestTestcase$ from the heuristic search is recycled.

A force directed algorithm is derived from force directed scheduling algorithms [16]. In difference to the previous greedy algorithms, the order in which the test cases are selected as candidates for elimination is not fixed a priori, but depends on the test cases that are still in the testsuite. We introduce a new function

$$timesCovered : PC \times Testsuites \rightarrow \mathbb{N}$$

to calculate how many test cases of a testsuite satisfy a partial coverage.

The *entry* function selects the test case with the lowest quality according to

$$quality(tc) = \frac{\sum\limits_{pc \in PC} \begin{cases} \frac{1}{timesCovered(pc, reducedTestsuite)-1} & : \; pc \in subPC(tc) \\ 0 & : \; else \end{cases}}{t_{exec}(tc) + t_{reset}}$$

The meaning of the formula is as follows: The numerator contains a metric for the assets achieved by the test case tc. The asset is the smaller the more frequent a partial coverage is achieved by other test cases in the testsuite. If tc is the last test case achieving a partial coverage, the asset is set to infinity, which maximizes its quality and prevents tc from elimination. In the denominator we have the test execution time of tc and the reset time. Thus faster test cases are favored.

Finally, we combine a search for test case generation and an optimization by a min-set-cover algorithm: $MinSetCover(search(model, PC), t_{reset})$.

4 Case Study: Robot Control

Tab. 1. Optimised testsuites generated for the robot control software by *EGRET*

component	RobotHardware	SensorModule	Planner (6 variants)	ControlThread	CycleThreadCheck	CycleThread	skPrControl	skPrInterface	RoboProgClient	GUI	RoboProgServer
number of configurations	27	10^4	216	10^7	256	10^{10}	10^7	10^8	10^{10}	10^5	10^7
number of partial coverages	1	7	3	14	5	38	12	7	32	16	19
testcase reduction (greedy) [%]	0	50	67	86	60	81	83	86	81	88	53
testcase reduction (heuristic) [%]	0	50	67	86	80	97	92	86	97	88	93
time reduction (greedy) [%]	0	50	69	86	60	81	82	83	79	76	54
time reduction (heuristic) [%]	0	50	69	86	80	92	88	83	93	76	92

The presented techniques have been implemented in our *E*xtendable *G*enerator for *E*fficient *T*estsuites (*EGRET*). *EGRET* imports UML models with some syntactic restrictions from Rhapsody for Java (from i-logix) in the XMI1.2 format. A model diagram as well as a statechart for each class is required. AND-states, method-calls which are not used for sending events, all other data types but bounded integers and booleans, and events with parameters are forbidden. However, we are working on a version supporting top-level concurrency, OR-states within the statecharts, the call of non-recursive functions and events with parameters. The exported system definition is instrumented and translated to timed automata. Coverage criteria, search strategies and min-cover-set-algorithms are plug-ins. Thus the tool can easily be extended. We are using UPPAAL as a model checker for test case generation. Some search strategies already use the possibility of a distributed execution of the model checker to lower test generation time even more. The traces which are output of the model checker and specify the test cases are translated into an XML-format which can be executed by a testdriver via a middleware on the components under test.

In the Collaborative Research Centre 562 ”Robotic Systems for Handling and Assembly”, a robot control software for parallel and hybrid kinematic machines has been developed. All components of the system have been modelled during the development process using sequence diagrams and later on statecharts [18]. The system consists of 16 components, with 27 up to $\sim 2 * 10^{10}$ configurations each, resulting in a state space of about $5 * 10^{93}$ states. The test generator is capable of generating conformance testsuites for all components and interoperability testsuites for pairs of communicating processes.

We applied *EGRET* to the statechart description of the robot control software, using different optimisation techniques. Table 1 shows, how the testsuite size can be reduced by a simple search combined with a bidirectional greedy algorithm and a heuristic search combined with a force directed greedy algorithm. The second approach reduces the test execution time up to 7% of the execution time of the unoptimised testsuite. The best results have been achieved in optimising the testsuites of large components, like the RoboProgClient, the RoboProgServer and the CycleThread. However, in case of smaller components, the results for the heuristic search and the simple search both combined with min-cover-set-algorithms are the same.

5 Conclusion

We presented an approach for automated model based testsuite generation. We considered a catalogue of test quality criteria, namely the test of system properties or interaction sequences, various coverage criteria, and mutant testing. We showed how to uniformly instrument state based models by adding variables or specific test drivers such that a model checker searching for a subgoal encoded as partial coverage will generate a trace that can serve as a test case for that subgoal. Thus, not only formal test quality criteria like coverages but also expert knowledge and existing tests in terms of sequence diagrams are integrated in an automated, formally founded test case generation smoothly.

Next we compared several heuristics for optimizing the test execution time without decreasing the test quality. We combined search algorithms, adding redundancy on the required coverages to a testsuite, with different min-set-cover algorithms that preserve the set of coverages but minimize the execution time.

Our experimental results on the case study are promising under three aspects: First, the test execution time could be significantly reduced. Second, the heuristics for optimization are efficient w.r.t. time and memory consumption such that our approach is applicable on medium sized real world case studies at least which is an significant improvement compared to other approaches. Third, our approach favors the generation of long test cases on which several subgoals (partial coverages) are tested. Thus, we avoid weaknesses w.r.t. the fault detection rate that were observed on other approaches to automated testsuite generation.

In future, we will investigate the interdependence between different strategies for automated model based testsuite generation and the ability to detect faults running the testsuite. First insights have been given in [7, 8], but a more systematic investigation can lead to valuable hints for what kind of systems which strategy can be recommended. Additionally, we plan to extend our work on mutant testing for state based models and investigate alternative heuristic optimization algorithms for testsuite generation like e.g. genetic algorithms.

References

1. Mücke, T., Huhn, M.: Generation of optimized testsuites for UML statecharts with time. In Groz, R., Hierons, R.M., eds.: TestCom. Volume 2978 of LNCS., Springer (2004) 128–143
2. Engels, A., Feijs, L., Mauw, S.: Test generation for intelligent networks using model checking. In Brinksma, E., ed.: Tools and Algorithms for the Construction and Analysis of Systems. (1997)
3. Rayadurgan, S., Heimdahl, M.: Coverage based test-case generation using model checkers. In: Intl. Conf. and Workshop on the Engineering of Computer Based Systems. (2001) 83–93
4. Hong, H., Lee, I., Sokolsky, O., Cha, S.: Automatic test generation from statecharts using model checking. In Brinksma, E., Tretmans, J., eds.: Workshop on Formal Approaches to Testing of Software (FATES). (2001) 15–30
5. Pretschner, A.: Classical search strategies for test case generation with constraint logic programming. In Brinksma, E., Tretmans, J., eds.: Workshop on Formal Approaches to Testing of Software (FATES). (2001) 47–60
6. King, K.N., Offutt, A.J.: A Fortran language system for mutation-based software testing. Software–Practice & Experience **21**(7) (1991) 685–718
7. Heimdahl, M.P., Devaraj, G., Weber, R.J.: Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In: Proceedings of the 8th IEEE International Symposium on High Assurance Systems Engineering (HASE), Tampa, Florida (2004)
8. Heimdahl, M.P., Devaraj, G.: Test-suite reduction for model based tests: Effects on test quality and implications for testing. In Wiels, V., Stirewalt, K., eds.: Proc. of the 19th IEEE Intern. Conference on Automated Software Engineering (ASE), Linz, Austria (2004)
9. OMG: Unified modeling language specification (2003) Version 1.5.
10. Larsen, K.G., Pettersson, P., Yi, W.: UPPAAL in a nutshell. International Journal on Software Tools for Technology Transfer **1**(1-2) (1997) 134–152
11. Diethers, K., Goltz, U., Huhn, M.: Model checking UML statecharts with time. In Jézéquel, J.M., Hußmann, H., Cook, S., eds.: UML 2002, Workshop on Critical Systems Development with UML. (2002)
12. Diethers, K., Huhn, M.: Vooduu: Verification of object-oriented designs using uppaal. In Jensen, K., Podelski, A., eds.: TACAS. Volume 2988 of Lecture Notes in Computer Science., Springer (2004) 139–143
13. Robinson-Mallett, C., Liggesmeyer, P., Mücke, T., Goltz, U.: Generating optimal distinguishing sequences with a model checker. In: A-MOST '05: Proceedings of the 1st International Workshop on Advances in Model-based Testing, New York, NY, USA, ACM Press (2005) 1–7
14. Sugeta, T., Maldonado, J.C., Wong, W.E.: Mutation testing applied to validate SDL specifications. In Groz, R., Hierons, R.M., eds.: TestCom. Volume 2978 of LNCS., Springer (2004) 193–208

15. Offutt, J., Pan, J., Voas, J.: Procedures for reducing the size of coverage-based test sets. In: Proceedings of the Twelfth International Conference on Testing Computer Software. (1995) 111–123
16. Paulin, P., Knight, J.: Force-directed scheduling for the behavioural synthesis of asics. IEEE Trans. on Computer-Aided Design **8**(6) (1989) 661–679
17. Garey, M., Johnson, D.: Computers and Intractability: A Guide to the Theory of NP-Completeness. Freeman and Company (1979)
18. Steiner, J., Diethers, K., Mücke, T., Goltz, U., Huhn, M.: Rigorous tool-supported software development of a robot control system. In: Robot Systems for Handling and Assembly, 2nd Colloquium of the Collaborative Research Center 562. (2005) 137–152