

A Comparison of Three Black-Box Optimization Approaches for Model-Based Testing

Teemu Kanstrén
VTT, Oulu, Finland
UofT, Toronto, Canada
teemu.kanstren@vtt.fi

Marsha Chechik
University of Toronto
Toronto, Canada
chechik@cs.toronto.edu

Abstract—Model-based testing is a technique for generating test cases from a test model. Various notations and techniques have been used to express the test model and generate test cases from those models. Many use customized modelling languages and in-depth white-box static analysis for test generation. This allows for optimizing generated tests to specific paths in the model. Others use general-purpose programming languages and light-weight black-box dynamic analysis. While this light-weight approach allows for quick prototyping and easier integration with existing tools and user skills, optimizing the resulting test suite becomes more challenging since less information about the possible paths is available. In this paper, we present and compare three approaches to such black-box optimization.

Keywords—model based testing; test automation; evaluation; test generation; optimization

I. INTRODUCTION

MODEL-BASED testing (MBT) [1] is a technique for generating test cases from a test model. As opposed to manual test design, where a test expert designs test cases one by one, in MBT the test expert designs a test model to represent the system behavior and uses an MBT generator tool to generate a set of test cases from it.

As the generator can potentially create a very large number of test cases for any non-trivial model, optimization approaches are often applied to choose which tests to include in the generated set. Many optimization solutions [2, 3, 4, 1, 5] use a custom notation combined with a specialized runtime environment to represent the test model, allowing for performing an in-depth white-box static analysis, e.g., symbolic execution, of the model. Tools such as constraint solvers can then be applied on this information to find test cases that yield short paths to reach given coverage targets [3, 6, 5]. Some use manually crafted scenarios to guide the generator towards specific paths [2].

In our work, we have aimed to provide light-weight solutions suitable for easy industry adoption and to support a fast iterative testing process through rapid prototyping of test models, test generation, and test execution, while still providing good test coverage and useful test results. We use a general-purpose programming language (Java) to represent our test models, allowing use of all language features, tools, libraries, and the standard (Java) virtual machine (JVM) runtime. This enables using existing development skills and toolsets such as test libraries and environments, integrated development envi-

ronments, and continuous integration systems, along with any features they provide. Our open-source test generator called OSMO Tester [7] has been successfully applied, together with industry partners, to test large scale real industry systems.

Our test model is as an executable program, executed in different ways by the generator to produce test cases. We allow use of different, evolving versions of the language platform (virtual machine) and features to create the model and run the generator on it. As the model can make use of third-party libraries and networked services, we assume no access to source code or even all binaries for analysis. Such limitations are common in practical settings (e.g. [8]). In this context, we cannot apply approaches based on white-box static analysis but rather rely on dynamic analysis and information available at runtime. We call this type of test generation *model-based black-box test generation*. While there has been extensive research into optimizing test generation using static analysis based approaches, little work exists in optimizing model-based black-box test generation.

In this paper, we describe three algorithms for optimizing test generation in this type of an environment. All of them are based on generating a large set of potential tests in parallel and picking the most optimal ones based on given coverage criteria. One is targeted at *online testing* where test generation and execution are interleaved. Two others are targeted at *offline testing* where the test set is first generated and later executed in a separate phase. We compare the strengths and weaknesses of the three algorithms and make usage recommendations.

The rest of the paper is structured as follows. Section II presents background on our modelling notation and test generator. It also defines how we assess achieved test coverage over the test model. Section III presents our optimization algorithms. In Section IV, we evaluate each algorithm individually in terms of achieved coverage, generation time and test length. In Section V, we compare the different approaches to each other. In Section VI, we compare our results with related work. We conclude in Section VII with a summary of the paper and discussion of future research directions.

II. BACKGROUND

To provide background for the following sections, we first briefly outline our modelling notation and model structure and describe our notion of test coverage, including how model elements are used to calculate coverage.

A. Modelling Notation

OSMO Tester uses a generic programming language (Java) as the modelling language. The models are executable and capture a set of possible test steps and how they can be combined to produce valid test cases. To generate test cases, the test generator executes different paths through the model which is often referred as a *model program* [9, 10]. In Figure 1, we illustrate our notation using a simple test model for a counter which can perform two functions: *increase* (the value of the counter by one) and *decrease* (the value of the counter by one). For illustration purposes, we will later use '+' to represent the increase step and '-' the decrease step.

In this model, the *system under test* (SUT) is represented by the *sut* variable. This example illustrates an *online* testing approach where the test steps are concretely executed against the SUT as they are generated. In an *offline* approach, the test steps yield a script which also includes the input for the SUT and the checks to perform against its states and output.

```

1: public class CounterModel {
2:   @Variable //annotation to identify interesting model state to generator
3:   private int value = 0;
4:   private Counter sut = new Counter();
5:   private Requirements req = new Requirements();
6:   @BeforeTest
7:   public void start() {
8:     value = 0;
9:     sut.reset();
10:  }
11:  @Guard("decrease")
12:  public boolean allowDecrease() {
13:    return value > 1; //when true, "decrease" is enabled
14:  }
15:  @TestStep("decrease") //enabled when above guard true
16:  public void decreaseCounter() {
17:    value--; //updates our model state
18:    sut.decrease(); //execute test step on SUT
19:    assertEquals(value, sut.value); //test oracle, check model vs SUT
20:    req.covered("decrease"); //user defined coverage requirement
21:  }
22:  @TestStep("increase") //has no guards, so is always enabled
23:  public void increaseCounter() {
24:    value++; //updates our model state
25:    sut.increase(); //execute test step on SUT
26:    assertEquals(value, sut.value); //test oracle, check model vs SUT
27:    req.covered("increase"); //user defined coverage requirement
28:  }
29:  @CoverageValue( public String zero() {
30:    return "" + (value == 0);
31:  }

```

Figure 1. Example counter model program.

The two basic model elements here are the methods annotated with *@TestStep* and *@Guard*. The *@TestStep* methods represent test steps that are executed by the test generator at different times to produce a test case. In MBT, these are also referred to as *actions* [11] and *action methods* [5]. Each test step invokes a function on the SUT, updates the model state, or checks the SUT state and output against the expected values (the test oracle). A sequence of these steps forms a *path* through the model, producing a test case.

To define the potential paths, i.e., the steps the generator can take in a specific model state, the test generator executes all *@Guard*-annotated methods (line 11 in Figure 1). These define rules for enabling test steps. When the guards for a step

become true, the step is enabled, and the generator can choose to take any of the enabled steps. The guards are matched to steps based on the names given as annotation parameters. For example, in the beginning *value* is 0 and thus the guard for *decrease* is false, meaning the test can only start with the *increase* step.

B. Specifying Coverage Values

In our example, the current value of the counter is stored in the model as the *value* variable. The annotation *@Variable* identifies this variable to be of relevant for the generator to track for coverage. To provide a test oracle, the *value* variable is constantly updated to match the expected value as result of actions executed against the SUT (lines 17 and 24). These actions are the test steps invoking the *increase* and *decrease* functionality of the actual SUT (lines 18 and 25). The test oracle compares the actual value in the SUT with the expected value in the model (lines 19 and 26).

We also define a set of additional terms for elements of the model when calculating our test coverage. We use the term *step-pair* to refer to two steps following one another in a test case. For example, a path of '+++' would have three unique step-pairs: '++', '+-', and '-+'.

Coverage *requirements* (lines 20 and 27) can be used to tag specific paths of interest through the model. Methods annotated with *@CoverageValue* annotation (line 29) can be defined to produce values to record as covered for specific paths. Typically, these record specific instances or combinations of interest for model state. In our example, it records whether the value 'zero' is covered by a path. As such functions are typically related to model state, we refer to them as *user defined state coverage*. When two different values are observed in a sequence inside a test path, the term *state-pair* coverage is used. State-pairs are recorded similarly to step-pairs but with the user defined state values.

The annotations, variable names and values, and method information are accessed at runtime using the standard JVM reflection support, maintaining the black-box quality of our approach.

C. Coverage Calculation

In order to evaluate model coverage achieved by the generated test cases, our generator keeps track of any coverage values it has observed during test paths. Users can give weights to the different model elements to focus coverage where interesting. For example, one can set some weights to zero to ignore them, or tune state weight lower to avoid large state-spaces taking over all others.

The model elements used for coverage calculation and their default weights used by our tool are given in Table 1. The default weights are based on our experience with various test models and related test generation.

Using *E* to represent a model element and *W* its weight, the formula to calculate the coverage score is:

$$\sum_{N=1}^7 (E_N * W_N)$$

N	Model Element	Default Weight
1	Variables	10
2	Variable values	1
3	Test steps	20
4	Step pairs	30
5	Requirements	50
6	User defined states	50
7	State-pairs	40

Table 1. Coverage elements.

That is, the number of unique values observed for each model element is multiplied by the weight for that element, and the sum of these forms the coverage score for a test case. For example, a test case TC1 with a path of ‘++-++’ would cover 1 variable (*value*), 3 variable values (1,2,3 for *value*), 2 steps (‘+’, ‘-’), 3 step-pairs (‘++’, ‘+-’, ‘-+’), 2 requirements (‘increase’, ‘decrease’), 1 user defined state (‘false’), and a single state pair (‘false-false’). The score of this path is thus $10*1+1*3+20*2+30*3+50*2+50*1+40*1 = 333$.

The coverage score for a test suite (all test cases together) is calculated by adding up all items in all test cases and then applying the formula. For example, suppose a test suite consists of TC1 defined above and a test case TC2 covering the path ‘++-++’. If TC2 was the only test case, its value would be $10*1+1*2+20*2+30*4+50*2+50*2+40*3 = 492$.

However, since TC1 is already in the suite, only the new elements added by TC2 are considered. These are a new step-pair ‘-’, a new user-defined state ‘true’ and two new state-pairs ‘false-true’ and ‘true-false’. Thus the suite score rises by $30*1$ (step-pair) + $1*50$ (state) + $2*40$ (state-pairs) = 160. The final coverage score for this test suite is then $333+160 = 493$. In general, our scoring function guarantees that the score of a test suite is independent on the order of adding test cases.

A common goal for testing is covering variance over important properties of the SUT [12]. The goal of our coverage algorithm is to provide a measure of overall model variance coverage for the optimization algorithms, which we expect to be designed to represent important aspects to test in the SUT.

III. OPTIMIZATION ALGORITHMS

In this section, we describe our optimization algorithms. First we describe the online optimization algorithm (Section 3-A), followed by the two offline algorithms (Sect. 3-B and 3-C).

A. Online Algorithm

Online testing interleaves test generation with test execution. Once the generator chooses a test step, it immediately executes it against the SUT; once the step has finished executing, it chooses the next step, etc. For us, online testing provides immediate test results and gives fast feedback to the test generation, modelling and evaluation process, and thus we want online test generation to be as fast as possible.

This type of real-time online test generation prevents us from doing optimization beforehand as we need to support cases where changes are made to the model, the generator is immediately invoked, and tests are generated and executed. To support this scenario, our online optimization algorithm explores sets of potential future steps while the previously chosen step is still executing.

The high-level algorithm is described in Figure 2. When the generator has chosen a step to execute (using the algorithm in Figure 2) but before it has executed it, it starts the exploration of the next step in parallel threads (or concurrently on a networked larger machine as discussed in [13]). It takes the sequence of steps so far executed (LS) for the current test case and the exploration depth (D) given by the user as input. The exploration depth defines how many steps the algorithm tries to look into the future in parallel to current step execution.

Input: current model instance and state CM used for concrete test generation, list of steps LS currently taken in test case, the exploration depth D.
Output: The next step to take.

1. execute all model guards on CM to construct the set of enabled steps ES
Set the set of potential future paths PP to empty set \emptyset .
2. for each step S in ES
3. create a new instance M of the model program
4. set M in exploration mode
5. execute all steps in LS on M to reach current test state for M
6. execute S on M to reach a new state NS
7. decrease D by 1
8. if $D > 1$, repeat from 2
9. else add this path for M to PP
10. set value for best path score B to 0, create new empty set of best paths SB
11. for each potential path P in PP
12. calculate coverage score CS for P
13. if $CS > B$, clear SB and set B as CS
14. if $CS == B$, add P to SB
15. return a random choice from SB as the next step to take

Figure 2. Algorithm for online optimization.

The starting point is the current concrete model instance CM (and its state) used to generate test cases. To explore the potential future paths, the algorithm starts with the set of enabled steps ES, that is the set of steps in CM where no guard returns false, as explained in Section II-A. For each step S in ES, a new instance M of the model program is created. This is set to exploration mode by invoking *@ExplorationEnabler* annotated methods on the model. This should, for example, replace a reference to a real SUT with a mock version that invokes no real functionality and thus has no visible side-effects when steps in M are invoked. The current set of steps LS is executed on this M to achieve the current generation state for M. This is repeated for each S, so that each S has its own instance of M that is in the same generation state.

For each of these instances of M, the associated S is invoked. If the exploration has at this point reached depth D, the coverage score for each M is calculated and that is the score that is given to this path. If D is not yet reached, all these executed paths for the different instances of M are taken as the new sets of steps LS and the algorithm starts from beginning for all these, with the value of D reduced by one.

In the end, all the final paths are taken and the highest scoring ones are collected. If there are several, the one that has the highest score fastest (in fewer steps) is taken. If several are still equal, one is chosen at random. The next unexecuted step on this path is given as a choice for the generator. In some cases, the algorithm still has to wait some time for the parallel exploration to finish. This can be mitigated by forming an “exploration buffer”. That is, exploring several future steps at a time when there is a chance and using that as a buffer.

B. Offline Algorithms

Our offline optimization algorithm has two different variations. One aims to optimize for a large test set covering a high variation of the test model elements, using the coverage formula presented in Section II.B. The other one aims to optimize for minimal test lengths to cover specific targets in the test model.

Greedy Variation Optimizer. The version targeting high variation is a form of greedy algorithm. It is described in Figure 3. The main arguments it takes from the user are the population size (PS) and the timeout value (TO). The number of parallel generators (P) to run can also be configured but defaults to the number of processing units for the system.

The greedy algorithm runs P versions of generators G in parallel. In Figure 3 This is represented as G_{1-P} meaning there are P different instances of test generators. The different ones are referred to as G_x , where x stands for one value from 1-P, or one instance of the generator. Each G_x is configured with the test generator configuration (GC) provided by the user and automatically populated with a unique randomization seed (to produce different test cases). Each G_x generates a given number of new test cases (PS). The new generated set of tests GS for G_x is merged with the existing test suite TS (which starts empty). Every T in TS is then iterated and highest scoring tests are added to the existing test suite TS for each G_x . First, the test T that has the highest coverage score in GS is added to TS and removed from GS. This process repeats until GS is empty or no T gets a positive score. Same procedure is done for each G_x generating a new GS and merging with TS.

Finally, once the overall generation timeout TO is reached or TS has not changed throughout the entire iteration, the process is stopped for that G_x . Once all G_x are finished, the TS for all G_x are combined to form the final optimized test set OS which is returned. This is done similar to creating the set for a single generator, starting with the highest scoring test in the overall set, followed by the test that adds most to this test (suite), and so on.

<p>Input: generator configuration GC, population size PS, timeout TO, degree of parallelism P.</p> <p>Output: Generated test suite OS</p> <ol style="list-style-type: none"> 1. create P instances of test generator as G_{1-P}, configured with GC 2. for each G_x, in G_{1-P} run the following in parallel 3. create empty test suite TS (\emptyset) 4. create unique randomization seed for G_x 5. use G_x to generate PS new test cases as the test set GS 6. add tests in TS to GS 7. clear TS, setting TS to empty set \emptyset 8. for each test T in GS 9. calculate added score AS for T when added to TS 10. add highest scoring T to TS and remove it from GS 11. if any T remains in GS with $AS > 0$, iterate from step 8 12. if timeout TO has been reached, stop generation with G_x 13. else if TS scores higher or is shorter than previous TS 14. continue from line 5 15. else stop generation with G_x 16. wait for all G_x to finish 17. merge results for all G_x using lines 8-11 as output set OS 18. return OS

Figure 3. Greedy algorithm for offline optimization.

Single Target Optimizer. The *single target* offline optimization approach, described in Figure 4, aims to generate a set of test cases where each coverage requirement is covered by a single test case of the shortest possible length. This can be useful if specific tests are needed but we want to have them generated from the model as opposed to manual scripting.

This algorithm starts by generating test cases as random walks through the model, using a set of P test generators G_{1-P} running in parallel. Each G_x is used to generate a given number PS of test cases. When a G_x finds a new test case T that covers a previously uncovered requirement R, it changes the global generator state to target finding R and to only allow the steps in T. T now becomes the *reference test*, called BT, for covering R. Upon finishing their iteration of generating PS tests, each G_x reconfigures itself with the new global generator state. This means that the goal of every G is to find a shorter path to cover R. The set of R to find can be given to the algorithm or it can pick them up from the model as it generates tests from it.

To further help the generators find potentially shorter paths, the global state is modified after each iteration to look for only those tests which are shorter than BT. Each G_x is configured to allow each available step one time less than in BT. If any G_x finds a new test T with a shorter path to R, this T becomes the new BT for R and all G_x reconfigure to target shortening this new BT. After test timeout TO is reached or the path cannot be shortened any more (it only has instances of one step), the final BT for R is added to the final output set OS.

Finally, the search is restarted with the goal of finding a new test for an uncovered R. The previously covered requirements are ignored at this point. The process is repeated until all requirements have been covered or suite timeout SO is reached.

<p>Input: generator configuration GC, population size PS, suite timeout SO, test timeout TO, degree of parallelism P.</p> <p>Output: Generated test suite OS with one test case for reaching each requirement R</p> <ol style="list-style-type: none"> 1. create P instances of test generator as G_{1-P}, configured with GC 2. create unique randomization seed for each G_x in G_{1-P} 3. while SO has not been reached 4. run each G_x in parallel to generate PS test cases as test suite TS 5. if any test T in any TS for any G_x reached a new uncovered R 6. set R as target to cover for each G_x 7. set best test BT for R to T 8. for each step S in BT 9. reconfigure all G_x to only allow the steps in (BT - S) 10. use G_x to generate PS new test cases TS2 11. if any test T2 in TS2 is shorter than BT, set BT to T2 12. iterate from line 8 until minimal BT achieved or TO is reached 13. add BT to final output test suite OS 14. iterate from line 1 until SO is reached or all requirements are covered 15. return OS
--

Figure 4. Single target algorithm for offline optimization.

IV. EVALUATION

In this section, we describe the evaluation of the algorithms. Due to space reasons, we cannot include all the details of all the test runs here. The detailed results are available in [14].

While we have experience with several industry models, we cannot describe these due to confidentiality reasons. Thus our evaluations use three publically available models. Two of these, a test model for a movie reservation system (ECinema), and for a GSM SIM card [1], are ported from the ModelJUnit MBT tool [15]. The third model was previously developed by us for a web application called iTrust, which is a role-based healthcare application [16]. While these are not actual industry models, in our experience the relevant complexity is similar in terms of relevant structural coverage elements of the model such as states, and test steps. The models are available as part of our tool website and repository [7]. Table 2 summarizes the sizes of these models in terms of their model elements (used in coverage calculations).

Model	Steps	Step-Pairs	States	State-Pairs	Reqs
ECinema	19	177	9	40	17
SIM	15	240	2.6k+	22k+	32
iTrust	40	800	47	1124	11

Table 2. Model sizes.

We used the default coverage weights described in Table 1 with one exception. For the SIM model we set the state weight to 5 and state-pair weight to 1 in order to avoid the huge state space taking over all other coverage criteria. This is a typical example of tuning the coverage weights per domain as discussed in Section II-B.

A. Online Algorithm

For this algorithm, we are interested in the improvements of coverage when compared to the baseline approach of random step selection and any delay it adds. That is, we are interested in the cost and benefit of exploring the next step(s) while the previous are executing.

To set our evaluation parameters, we did an initial scalability study in which we found that test length of 100 steps and a depth of 2 were the most suitable parameters. Thus we used these in the online algorithm evaluation.

Coverage. To evaluate the algorithm in terms of coverage, we ran the test generator against each of the three test models. We did this with the random algorithm, and with the exploration algorithm using depths of 1 and 2. We repeated each of these experiments 100 times and collected the minimum, maximum and average achieved coverage for each algorithm in each configuration for each model. The results indicate that exploration algorithm generally outperforms the random selection for all our coverage criteria.

Timing. To evaluate the timing aspect of the algorithm, we used the iTrust system as a test subject as it was tested through the web-based graphical-user interface (GUI), which in our experience is a good candidate platform for parallel optimization due to test execution delays. The experiment was run on our laptop system, also running the SUT, including the database server, and the webserver. Test execution used the Selenium WebDriver (Chrome) component, allowing control of an actual browser to simulate a test user using the application. Thus, the flow is the same as with an actual user, with all the requests going through the whole SUT from the browser to the backend database and back.

As random selection is practically instant, it was used as a baseline for comparison. To briefly summarize the interesting parts, exploration at depth of 1 (Expl-D1) is in all cases faster than the execution of the concurrent test step (Random). Exploration at depth of 2 (Expl-D2) is in most cases much slower than the step execution time. This means that, for our setup, the depth of 1 is very reasonable, while depth of 2 is slower, although it does also achieve a much higher exploration score than depth 1.

Our goal in this study was also to evaluate the general feasibility of the approach to achieve near real-time test generation and execution. Here, this is true for depth 1. With optimizations and faster computing resources (e.g., [13]) we believe this is within reach for depth 2. Of course, this also largely depends on concrete test execution speed, which may give us more or less time for exploration. In our experience, long delays are common in many cases, specifically, with GUI-based testing.

B. Greedy algorithm

For this algorithm, we were interested in evaluating how well it covers variation over the test model structure. We were also interested in the time it takes to achieve the coverage and length of the resulting test cases.

Coverage. To evaluate the algorithm in terms of coverage, we again used all three of our test models. Comparing the results against the baseline random selection algorithm and the online optimization algorithm, Greedy generally outperforms the other two over long term. However, while Greedy performs much better than exploration in the long run, it may sometimes perform slightly worse for some properties.

Figure 5 shows the overall coverage score evolution for one run of each algorithm as new tests are added to the test suite. The coverage score shown is the overall score of the test suite as shown in Section II-I.C. The obvious observation is how random selection yields much lower coverage score than the other algorithms. What is not so clearly visible is how the exploration algorithms score slightly higher than greedy in the beginning, with greedy surpassing depth 1 at test 30 and depth 2 at test 100.

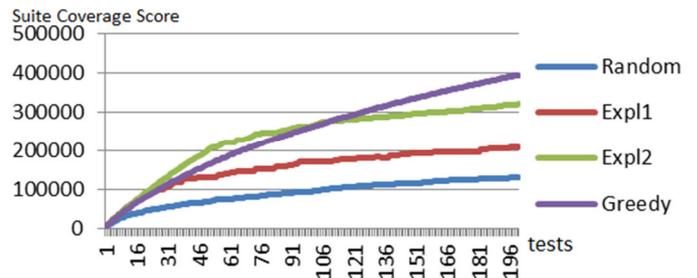


Figure 5. Coverage score evolution over 200 tests.

Timing. Unlike the online exploration algorithm, the greedy algorithm takes time to start and build the test suite for later execution. In this case running the greedy algorithm with our desktop configuration takes about 165 minutes. This is the delay one would have to wait before starting to execute the tests. Compared to the online exploration algorithm, the online version has no delay in the start and the overall generation time for the exploration at depth of 1 is about 20 minutes and for depth 2 about 105 minutes.

C. Single Target Algorithm

To evaluate the single target algorithm, we started by investigating the limits of the algorithm. To do this, we a model with 10 test steps, each always enabled. Thus the probability of taking any one of them with random choice is 10%. It has one requirement to cover, and the length of the path that needs to be taken to cover this requirement can be configured by the *target* parameter. Different values for this parameter were then applied with tests of different length.

The results indicated that the algorithm could always reduce the path to the shortest in all these configurations. Additionally, we used binomial distribution to verify these results. These calculations matched our experiments and thus confirmed the algorithm is useful for such model requirement coverage optimization. This also shows how this type of probability analysis can be used to tune the search parameters when we have an idea of the complexity of paths we need.

To further evaluate the performance, we used the SIM model as a test subject. It had a set of 35 coverage requirements defined by its original authors, and our target was to cover these with one test each. The results indicated that three requirements were not covered at all. In order to find out the reason, we manually analyzed the test model for the paths to reach these requirements. We found that due to complex ways the different test steps and model variables interact, it was not possible to reach those requirements in any way. While we are not the authors of the model, we assume that this is the result of model evolution where the requirements have not been rechecked, perhaps due to the complexity of this process. In any case, such knowledge in itself is valuable to better understand the test model and the generated test cases.

As for the 32 covered requirements, we found that the algorithm did find the minimal path to reach each of these requirements in the model. The results are good but we also realize that there can be more complex cases where the paths are more difficult to minimize due to dependencies between model steps and state variables. We leave further exploration of such models for future work. Again, we remind that the detailed results are available in [14].

V. DISCUSSION

Out of the three algorithms that we have presented, the online optimizer and the greedy offline optimizer target similar goals. Both try to optimize coverage for a variation of chosen parts of the model for each test case and the overall test suite. The single target offline optimizer is different in trying to generate a set of test cases where each one reaches a specific path requirement. It is not looking for the overall variation using our coverage score but rather for a specific set of test cases, each with a minimal set of steps for reaching a given requirement.

As shown in Figure 5, the online exploration approach gains coverage faster in the beginning but loses to greedy over the long term. This is due to the online algorithm finding many uncovered steps, states and their combinations in the beginning but lacking vision in later test cases for how to achieve a state from which it can again gain more coverage score. It is also due to the states being distributed more after the initial set has been covered. The greedy version, on the other hand, selects

from a large set of random tests, where the initial tests may not be as good as those in the online exploration version but where over time it finds more uncovered coverage elements in the larger set of random tests. For generating an overall regression test suite, a combination of these could be useful.

Overall, we prefer to use the online exploration algorithm when working on models for systems where test execution takes non-trivial time. One example is when we are evolving the models and trying to determine the impact of the modification on test results. In these cases, the algorithm quickly produces a variation over different parts of the model. If we want to focus this variation on specific parts of the model, we use scenarios to guide the generator to only consider these parts and the algorithm to produce a variation over these chosen parts. Scenarios are a way to tell the generator to ignore some steps completely, use a specific sequence to start every test, and to limit the number of times a test can contain some steps.

If we need the ability to execute large test suites, we have found the greedy or even the random approach to work better. When the tests execute fast (e.g., testing middleware or application logic), it is possible to run a large test set fast even on a single multi-core machine. In such a case, in the time it takes for the optimization approaches to produce the test set, the random selection and its online execution have very likely already achieved a high coverage score and also covered additional combinations. For slower to execute cases such as GUI-based testing, some of these benefits may also be achievable with large-scale testing tools such as Selenium Grid (e.g., overnight or over the weekend using a large set of machines) when we want to perform very extensive test runs.

One of the main considerations for us is the fast evolution of the test models. Modern software development is fast paced and changes are frequent, especially with agile software development practices. In such environments, we prefer the online approach. This allows us to version control only the test model (as opposed to thousands of tests), generate and execute the tests, modify the model and repeat. However, we also find it useful to generate an offline test suite for cases where we want a specific set covered every time, e.g., one larger set executed to provide a high overall model coverage once the external test interfaces have sufficiently stabilized.

This also brings us to the single target optimization algorithm. While the larger test suites generated by the greedy and online optimization algorithms generally also cover the requirements defined in those models, and in many cases a single test case covers many requirements, there can still be value in specific test cases. For example, they are useful to show management and domain experts how specific test requirements are covered by a concise set of test cases, or how specific paths through the model are formed by the generator. They can also be used as a smaller set of offline test cases to support a larger online test generation and execution process.

Our results indicate that using the single target algorithm, coverage requirements can often be more easily found by generating much longer test cases that the expected required length to cover the requirements, and then having the tool minimize them. As noted, the probability calculation presented in Section IV-C can be a useful tool to set these parameters.

One advantage of online testing compared to its offline counterpart is the ability of the former to adapt to responses and state changes of the SUT on the fly [1], e.g., when testing a non-deterministic system. While the algorithms we have presented mostly assume a deterministic test response, the online version can also adapt to non-determinism, with the exploration becoming the estimation of future paths, and each estimate is re-calculated for each new step.

As shown in Figure 5, the optimizations help achieve a more diverse coverage according to the defined score weights when compared to the random choice. We also found it useful, in practice, to add random variation to the generated test cases, to avoid expert bias, i.e., cases not considered by the human expert but exercised in practice. We find that the generation and optimization approaches based on our diverse coverage score (greedy and online look-ahead) work well to achieve such goals. Practically, they focus on achieving the given model coverage criteria and interleave these with elements of randomness in different parts. That is, the result may not be fully optimized to produce the smallest possible test sets, but generally, we prefer this over too aggressive optimization. This provides a good tradeoff between generating huge random test sets and only generating manually defined specific test sets, which do not make use of the large scale capacity of test generators and are subject to expert bias in what is tested.

VI. RELATED WORK

Test optimization in MBT has been a popular research topic with various tools and approaches being implemented. Tools such as Conformiq [3], Uppaal [4], and Spec Explorer [5] use customized modelling languages, and static analysis-based approach with symbolic execution and constraint solving to optimize test sets. Mostly these approaches target offline test generation, while some work [6, 4] has also made use of two-phased test generation where static analysis is first performed and the resulting information is used to aid in online test generation. In contrast, our approaches are targeted to cases where such static analysis is not available due to the complexity of the modeling language and environment.

Various approaches using general-purpose programming languages for modelling have also been presented. Model programs similar to ours are used in MBT tools such as Spec Explorer [5], PyModel, NModel [10], and ModelJUnit [1]. All of these tools use variations of random search for test selection. Spec Explorer, PyModel and NModel support guiding test selection through user defined scenarios which slice the model to focus test generation around the specific parts of the model. Similar scenarios are also supported in our generator, and these can be used together with the optimization approaches we presented to focus testing on specific model parts.

A black-box optimization approach for test generation from models expressed using general purpose modelling languages is also available with ModelJUnit [15]. It supports executing the test model in a simulation mode as a pre-analysis phase before starting the actual test generation. Possible sequences of steps observed in these runs are collected and used to guide the actual online test generation in the following phase. However, this approach does not consider the state of the model, which defines what paths are available, and thus the set of paths as-

sumed by the optimization analysis are different from the actual ones available during generation. The difference to our approaches is that we do not use a separate pre-analysis phase and produce accurate results where the exact paths and impacts on coverage are known.

Regression test selection, minimization and prioritization are topics related to optimizing test suites [17]. These typically consider the optimization in terms of different ways to cover the SUT implementation [17]. In this paper we have discussed optimization of test model coverage using a more varied set of coverage criteria at a higher abstraction level. However, investigating ways to use works such as described in [17] together with our approach is an interesting topic for future work.

The techniques presented in this paper can also be viewed as a multi-objective test optimization problem. We aim to cover the items defined for the scoring function in Section II, which can be seen as a fitness function in the terms of search-based testing (see e.g., [18]). The algorithms we apply for coverage score optimization can be seen as a dynamic multi-object optimization algorithm for the test suite. Various approaches in search-based optimization have been applied before [18] but none in our knowledge to model-based test generation with black-box constrains. In fact, we are not aware of previous work in MBT for optimizing a set of coverage criteria as diverse as the one we optimize in this paper.

A multi-objective approach to test suite optimization for product lines is presented in [19], targeting objectives such as cost of test case, cost of test requirement and product variants. Test coverage is considered as single coverage requirements on the test model as opposed to our extensive model variation coverage support. An interesting aspect to integrate with our work could be the association of cost to certain test targets as part of the coverage score function.

In software testing and verification, various combinations of static analysis and (random) test data generation have been used. For example, directed automated random testing (DART) [20] combines static analysis and observations about the running system with random inputs to guide it towards new paths. Java PathFinder (JPF) [21] enables generating test paths based on a combination of symbolic and concrete executions. While these are different testing approaches than MBT, combining the information from these types of different sources with test generation (similar to [6]) and the approaches presented in this paper could be an interesting future research topic. While it may be prohibitive in terms of wait time to perform such extensive pre-analysis, a possible approach could be to perform this as a background process during modelling similar to what is described for executing unit tests in [22].

In general, a lot of work in automated test generation targets the traditional code coverage as a test target. When additional test coverage targets are considered, these are typically specific coverage requirements such as labels on the test models [23]. In addition to the optimization algorithms we presented in this paper, the coverage scoring method itself extends these with wider generic model coverage criteria (the model structure elements), supported by domain-specific criteria (user defined state and variables).

Random testing has been shown to work well in various situations [24, 25], and some criticism has been voiced on the effectiveness of guided random testing approaches when the same coverage can be achieved by simply executing a large set of random test cases in the same time [25]. We share this view in noting that executing a very large set of random tests should yield the same or even higher coverage over time as our optimization approaches do. The aim of our approaches is to practically choose a reasonable subset of such a larger set under different use cases.

VII. CONCLUSIONS

In this paper, we presented three different approaches to test optimization in black-box model-based testing. We then evaluated their performance and provided comparisons on the strengths and weaknesses of each. Our evaluation highlights the benefits of these different approaches over the traditional random choice and their usefulness in different contexts. The online version works well to provide added coverage when prototyping slow to execute test cases. The greedy offline optimizer gives a test suite for higher overall model coverage. The requirements targeting optimizer can help provide specific test cases where useful. Combining potential benefits of these approaches with static analysis where possible would be an interesting future research topic. Domain-specific applications of these approaches, including customizations of algorithms and modelling languages, are also interesting future topics.

VIII. REFERENCES

- [1] M. Utting and B. Legeard, *Practical Model-Based Testing: A Tools Approach*, Morgan Kaufman, 2006.
- [2] W. Grieskamp, N. Kicillof, K. Stobie and V. Braberman, "Model-Based Quality Assurance of Protocol Documentation: Tools and Methodology," *Journal of Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 55-71, 2011. DOI: 10.1002/stvr.427
- [3] A. Huima, "Implementing Conformiq Qtronic," in *Testing of Software and Communicating Systems*, 2007.
- [4] M. Mikucionis, K. Larsen and B. Nielsen, "T-Uppaal: Online Model-Based Testing of Real-Time Systems," in *19th International Conference on Automated Software Engineering (ASE)*, 2004. DOI: 10.1109/ASE.2004.1342774
- [5] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann and L. Nachmanson, "Model-Based Testing of Object-Oriented Reactive Systems with Spec Explorer," *Formal Methods of Testing*, pp. 39-76, 2008.
- [6] D. Ahman and M. Käärmees, "Constrain-Based Heuristic Online Test Generation from Non-Deterministic I/O EFMSs," in *7th Workshop on Model-Based Testing*, 2012. DOI: 10.4204/EPTCS.80.9
- [7] T. Kanstrén, "OSMO Tester Home Page," February 2013. [Online]. Available: <http://code.google.com/p/osmo>. [Accessed February 2013].
- [8] A. Groce, A. Fern, J. Pinto, T. Bauer, A. Alipour, M. Erwig and C. Lopez, "Lightweight Automated Testing with Adaptation-Based Programming," in *IEEE International Symposium on Software Reliability Engineering*, 2012. DOI: 10.1109/ISSRE.2012.1
- [9] M. Veanes, C. Campbell, W. Schulte and N. Tillmann, "Online Testing with Model Programs," in *ESEC/FSE-13*, 2005. DOI: 10.1145/1095430.1081751
- [10] J. Ernits, R. Roo, J. Jacky and M. Veanes, "Model-Based Testing of Web Applications using NModel," in *Testing of Software and Communication Systems*, 2009.
- [11] M. Utting, A. Pretschner and B. Legeard, "A Taxonomy of Model-Based Testing Approaches," *Software Testing, Verification and Reliability*, vol. 22, no. 5, pp. 297-312, 2012. DOI: 10.1002/stvr.456
- [12] J. A. Whittaker, *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*, Addison-Wesley, 2009.
- [13] T. Kanstren and T. Kekkonen, "Distributed Online Test Generation for Model-Based Testing," in *Asia Pacific Software Engineering Conf.*, 2013. DOI: 10.1109/APSEC.2013.43
- [14] T. Kanstrén and M. Chechik, "A Comparison of Three Black-Box Optimization Approaches for Model-Based Testing," 5 June 2014. [Online]. Available: http://www.kanstren.net/appendix/ATSE2014_full.pdf. [Accessed 5 June 2014].
- [15] M. Utting, "The ModelJUnit Model-Based Testing Tool," 2009. [Online]. [Accessed 17 May 2013].
- [16] North Carolina State University, "iTrust: Role-Based Healthcare," 2013. [Online]. [Accessed 17 May 2013].
- [17] S. Yoo and M. Harman, "Regression Testing Minimization, Selection and Priorization: A Survey," *Software Testing, Verification and Reliability*, vol. 22, no. 2, pp. 67-120, 2012. DOI: 10.1002/stvr.430
- [18] P. McMinn, "Search-based testing: Past, present and future," in *3rd International Workshop on Search-Based Software Testing*, 2011. DOI: 10.1109/ICSTW.2011.100
- [19] H. Baller, S. Lity, M. Lochau and I. Schaefer, "Multi-Objective Test Suite Optimization for Incremental Product Family Testing," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2014.
- [20] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed Automated Random Testing," in *Programming Language Design and Implementation (PLDI)*, 2005. DOI: 10.1145/1064978.1065036
- [21] C. S. Pasareanu and N. Rungta, "Symbolic Pathfinder: Symbolic Execution of Java Bytecode," in *Automated Software Engineering (ASE)*, 2010. DOI: 10.1145/1858996.1859035
- [22] D. Saff and M. Ernst, "Reducing Wasted Development Time via Continuous Testing," in *Proc. Int'l. Conf. on Software Testing and Analysis (ISSTA)*, 2003.
- [23] S. Bardin, N. Kosmatov and F. Cheynier, "Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2014.
- [24] I. Ciupa, A. Pretschner, M. Oriol, A. Leitner and B. Meyer, "On the Number and Nature of Faults Found by Random Testing," *Software Testing, Verification and Reliability*, vol. 21, no. 1, pp. 3-28, 2009. DOI: 10.1002/stvr.415
- [25] A. Arcuri, M. Z. Iqbal and L. Briand, "Random Testing: Theoretical Results and Practical Implications," *IEEE Transactions on Software Engineering*, vol. 38, no. 2, pp. 258-277, 2012. DOI: 10.1109/TSE.2011.121