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 solutions 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 a set of test cases 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, various optimization approaches are often applied to choose which test cases 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 testing 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, networked services, and the standard (Java) virtual machine (JVM) runtime. This enables using existing development skills and toolsets such as test libraries and environments, integrated development environments, and continuous integration systems, along with any features they provide. Our open-source test generator called OSMO Tester [8] has been successfully applied, together with a number of 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 language 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. [7]). Thus, we cannot apply approaches based on white-box static analysis in this context 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. 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 at each point the generator can choose to take any of the enabled steps. The association of guards to steps is based on matching the names given as parameters to the annotations. 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 '-+'.

The user can also define their own paths of interest and functions to give values that the generator should track for coverage purposes. 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 return String 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 the sequence of test steps it has taken, and any coverage values it has observed during these steps. Using MBT terminology from [11], we support data coverage (state values and their pairs, model variables), model structure coverage (test steps and their pairs), and model requirements coverage.

Users can guide the tool to prioritize coverage of specific model elements according to current testing needs. This prioritization is achieved by changing weights of the different model elements. For example, one can focus on specific model elements by zeroing the other weights or tune state weight lower

for a model with a large state-space to limit focus on state at the expense of other criteria.

The model elements used for coverage calculation and their default weights used by our tool are given in Table 1. The default weights have been determined based on our experience in working with different test models and generating test cases for those in order to achieve overall high coverage.

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.

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

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

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 generated up to current point) 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, is value would be 10*1+1*2+20*2+30*4+50*2+50*2+40*3 = 492.

The difference to TC1 adding a new step-pair '--', which also leads to value zero for the counter variable and thus produces one new state 'true' and two new state-pairs 'false-true' and 'true-false'. However, since TC1 is already in the test suite, adding TC2 to it only increases the score of the suite by the values not already covered by any test in the suite. These are the new step-pair and the new state-pairs. 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. If TC2 was added to the test suite first, adding TC1 would only increase the suite score by 1 as the only element value not covered by TC2 in TC1 is the counter variable value 3. In general, our scoring function guarantees that the score of a test suite is independent on the order of adding test cases.

III. OPTIMIZATION ALGORITHMS

In this section, we describe our optimization algorithms. We first describe the online optimization algorithm (Section 3-A), followed by the two offline algorithms (Sections 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. In our experience, this is suitable for a set of systems where executing a test step takes a non-trivial amount of time, such as testing web- or smartphone-applications through a graphical user interface (GUI). By making use of such delays, the algorithm can help optimize the online test coverage in parallel with test case execution.

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 [12]). 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 Ø.
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.

Once the execution of the previous step has finished, the choice given as the next step is the result of this parallel exploration. If the exploration finished before the actual execution of the previous step, the choice is immediate as the result is available. Otherwise, the algorithm waits for the parallel exploration to finish. This may incur some extra delay, which can be mitigated by forming an "exploration buffer". This refers to starting the exploration for the following step after the one that was explored already, in cases where the exploration finishes before the concrete execution of a parallel step does.

For example, assume we start concrete execution of step 1 and exploration of step 2 at the same time. If step 2 finishes exploration before step 1 finishes executing, exploration can proceed to step 3. This helps even the differences between steps that take different times to execute and explore.

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 $G_{1\text{-}P}$ 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.

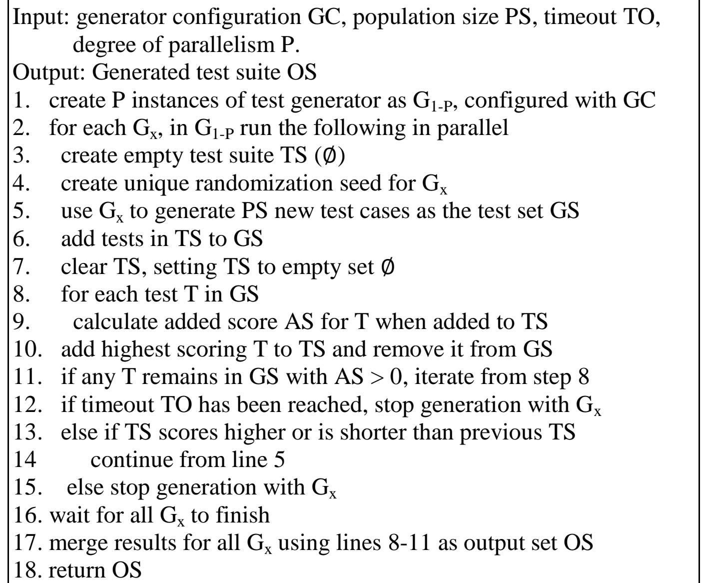
Input: generator configuration GC, population size PS, timeout TO, degree of parallelism P.
Output: Generated test suite OS
1. create P instances of test generator as $G_{1\text{-}P}$, configured with GC
2. for each G_x, in $G_{1\text{-}P}$ run the following in parallel
3. create empty test suite TS (∅)
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 ∅
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 5, 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 $G_{1\text{-}P}$ 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

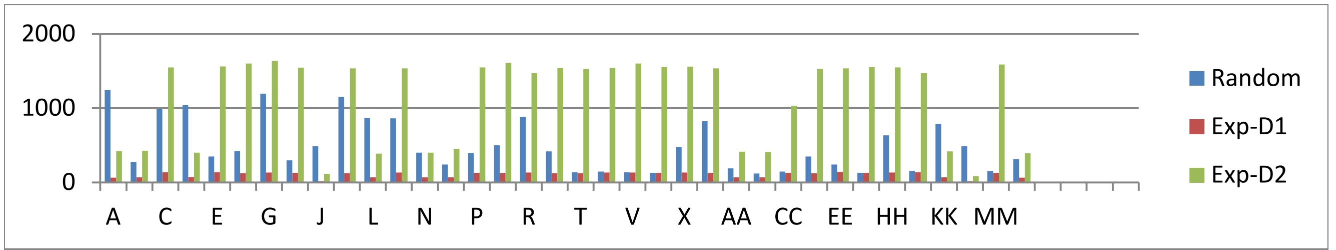

Figure 4. Exploration time vs execution time.

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.

Input: generator configuration GC, population size PS,
suite timeout SO, test timeout TO, degree of parallelism P.
Output: Generated test suite OS with one test case for reaching each requirement R

1. create P instances of test generator as $G_{1\text{-}P}$, configured with GC
2. create unique randomization seed for each G_x in $G_{1\text{-}P}$
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 5. Single target algorithm for offline optimization.

IV. EVALUATION

In this section, we describe the evaluation of the algorithms. Two different systems were used in the evaluations. One is a laptop with dual-core Intel Core i5 2.67GHz CPU. With hyper-threading, it has 4 virtual processing units, and our evaluations on it run 4 tasks in parallel. The JVM is configured with 2GB maximum heap size out of 4GB physical RAM. The second system is a desktop with quad-core Intel Core i7 2.8GHz CPU. With hyper-threading, it runs 8 parallel tasks. The JVM is configured with 20GB heap out of max 24GB.

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 [13]. The third model was previously developed by us for a web application called iTrust, which is a role-based healthcare application [14]. 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 [8].

Table 2 summarizes the sizes of these models in terms of their model elements (used in coverage calculations). The number of step-pairs also considers the start of test generation as a separate step, thus increasing the number of pair count by the number of steps. For example, SIM has 15^2+15 (225+15=240) step-pairs.

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. Relative to the other model elements, the SIM model had a large number of possible state and state-pair values. Thus we set its 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.

Model	Alg.	Steps			Step-Pairs			States			State-Pairs			Requirements		
		Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
ECin.	Rand	16	19	18,7	143	163	156	9	9	9	35	37	36,9	16	17	17
	Expl-1	18	19	19	154	168	161	9	9	9	37	37	37	17	17	17
	Expl-2	19	19	19	169	172	172	9	9	9	37	37	37	17	17	17
SIM	Rand	15	15	15	240	240	240	591	772	684	2562	2969	2770	30	32	30,1
	Expl-1	15	15	15	240	240	240	1619	1985	1799	4396	4998	4639	32	32	32
	Expl-2	15	15	15	240	240	240	2464	2670	2562	7111	7791	7413	32	32	32
iTrust	Rand	40	40	40	463	573	528	40	47	46,6	277	519	383	9	11	10,7
	Expl-1	40	40	40	649	800	762	45	47	46,9	410	685	528	9	11	10,9
	Expl-2	40	40	40	643	777	737	46	47	46,7	707	906	804	11	11	11

Table 3. Model coverage for 100 runs.

Metric	ECinema			SIM			iTrust		
	Min	Max	Avg	Min	Max	Avg	Min	Max	Avg
Tests	11	14	12,4	2311	2423	2362	55	61	58
Length	618	824	714	126348	131987	128722	3568	4011	3784
Steps	19	19	19	15	15	15	40	40	40
Step-Pairs	171	177	175	240	240	240	797	800	799
States	9	9	9	2527	2607	2571	46	46	46
State-Pairs	40	40	40	21513	22375	21916	1074	1078	1077
Reqs	17	17	17	32	32	32	1	1	1
Time(s)	6	14	9	5166	10498	7564	18	39	25

Table 4. Model coverage for 100 runs with greedy algorithm.

A. *Online Algorithm*

For this algorithm, we are interested in the improvements of coverage when compared to the baseline approach of random step selection. We are also interested in the time it takes to explore each step compared with executing the previous step. 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 using the three models with 10, 20 and 30 steps, no guards, and exploration depths of 1, 2, and 3. This means that the algorithm always had S^D steps to explore at every point (which is also the worst case complexity of the algorithm), ranging from 10^1 to 30^4, or 10 to 27000 steps. To set a realistic test length parameter, we ran all these combinations with test length up to 500 and measured the time for each length. As the exploration algorithm needs to repeat the current steps in the current test case for every explored future step, long test cases can cause it to slow down.

We found that at around 100 test steps with a model of 20 steps and depth of 2, the average time was around 200 milliseconds, which in our experience is a reasonable time for many GUI testing environments to take to execute a real step. For depth of 3, the time to explore a step for a model of this size was close to three seconds. Thus we chose to use depth of 2 as a realistic maximum depth of exploration. We also set 100 as our test length parameter since test cases observed in real projects were seldom longer than that.

Coverage. Table 3 shows the coverage results for our actual evaluation test models. In the left column, Rand refers to a random choice algorithm Expl-1 to our exploration algorithm with depth of 1, and Expl-2 with depth of 2. Each of these was run 100 times with different randomization seeds to produce 100 different test suites. For each algorithm, the table shows the minimum, maximum and average number of each elements these 100 test suites covered.

From this, we can see that the exploration algorithm outperforms the random selection for all our coverage criteria. An interesting detail is how the iTrust Expl-2 case achieves lower step-pair coverage than iTrust Expl-1. We speculate this is due to the algorithm putting more focus on covering a much higher number of state-pairs, which is due to the two having the same default weights (step-pairs and state-pairs).

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, which includes a MySQL database, a Tomcat webserver, and the iTrust application itself. 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.

The results are visualized in Figure 5. Random refers to the time it takes to choose a step and execute it using the random step selection algorithm. As the random choice is practically instantaneous, we use it as a baseline comparison to evaluate the overhead of the parallel exploration. Exp-D1 refers to exploration time for the next step while that step is running, using the exploration depth 1. Exp-D2 refers to the same with depth 2. For the sake of space and readability, four of the highest spikes (6s, 4s, 3s and 3s) are not shown in the figure. In all these four cases, the exploration time was much lower than test execution.

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 that 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., [12]) 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.

As shown in Table 3, depth 1 already gives a good increase in coverage over random choice. As shown in Figure 5, depth 1 also adds no extra delay to the test execution cycle, making it virtually free while improving coverage. We have already used this algorithm in practice as a quick prototyping aid for working with the different models, combined with other tools such as scenario definitions.

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. Table 4 shows the coverage results for the greedy algorithm in similar format to Table 3. Comparing these tables, Greedy outperforms Random and Exploration results in most cases. The exception is that in a few runs, the number of covered step-pairs for ECinema and states for SIM is lower for Greedy than the maximum achieved by some exploration run. That is, while Greedy performs much better than exploration in the long run, it may sometimes perform slightly worse for some properties.

Figure 6 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-II.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.

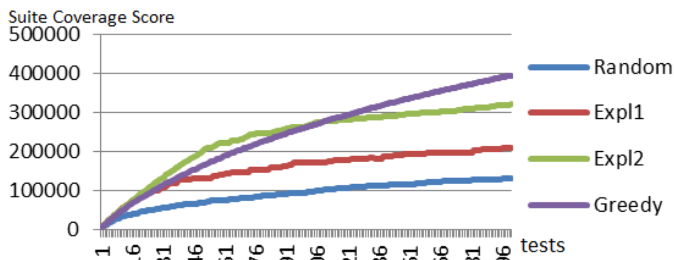

Figure 6. Coverage score evolution over 200 tests.

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 created the model shown in Figure 7. This model has 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.

Table 5 summarizes the results, with the *Tgt* column showing the value of the *target* variable and the *Len* column refering to the configured length of test cases generated. The length of the initial path for a requirement as found by the algorithm is given in the *Path* column. The final reduced path was always equal to *Tgt* column, which is also the shortest possible path. This is due to the algorithm reducing the number of steps experimentally at different iterations and finally reaching the only required steps for the requirement path (step 1 in Figure 7).

```
public class Model {
  private final int target;  //the given coverage target
  private int x = 0; //counter

  public Model(int target) {this.target = target;}

  @TestStep
  public void step1() {
    x++; //we have to hit this line target times to cover the requirement
    if (x == target) req.covered("target found");
  }

  //steps 2-10 are similar to this, bringing probability of taking step 1 to 10%
  @TestStep
  public void step2() {}…
}
```

Figure 7. Evaluation model.

Tgt	Len	Path	%	Tgt	Len	Path	%
5	50	5	1	30	50	-	0
5	100	5	1	30	100	70	0.02
5	200	5	1	30	200	76	1
10	50	10	1	35	50	-	0
10	100	10	1	35	100	-	0
10	200	10	1	35	200	106	1
15	50	15	1	40	50	-	0
15	100	15	1	40	100	-	0
15	200	15	1	40	200	150	1
20	50	-	0.02	45	50	-	0
20	100	27	1	45	100	-	0
20	200	27	1	45	200	-	0.16
25	50	-	0	50	50	-	0
25	100	46	1	50	100	-	0
25	200	52	1	50	200	-	0

Table 5. Single target algorithm performance.

Additionally, we used binomial distribution to calculate the probability (the *%* column in Table 5) of hitting a 10% chance (as for step 1 in Figure 7) *Tgt* times for *Len* experiments for each row in Table 5. This gives us the theoretical probability of finding a path to cover the requirement on each row. As can be seen in the table, the results match the calculated probability. For example, with *Tgt* 20 and *Len* 50, there is a probability of 2% to find a path. In this case it is not found. Similarly, with *Tgt* 30 and *Len* 100 there is a probability of 2% to find a path, and in this case we happen to find it. This shows how this type of probability analysis can be used to tune the search parameters when we have some idea of the complexity of paths we need, and letting the algorithm optimize it once found.

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. Table 6 shows the results for each of these requirements using a test length of 50 steps for generation and random choice as the initial step selection. *Req* is the identifier of each requirement, and *Start* is the length of the first test case the random generator produced that reached that requirement. *Steps* is the length of the final test case computed by our algorithm for that requirement.

The table shows 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.

Req.	Steps	Start	Req.	Steps	Start	Req.	Steps	Start
1.	5	17	14.	-		27.	1	3
2.	2	16	15.	-		28.	2	31
3.	1	17	16.	4	32	29.	1	3
4.	3	50	17.	2	10	30.	1	8
5.	1	7	18.	1	23	31.	3	17
6.	11	34	19.	4	24	32.	-	
7.	2	6	20.	2	34	33.	4	39
8.	1	12	21.	5	19	34.	4	38
9.	10	26	22.	2	24	35.	3	49
10.	1	4	23.	1	12			
11.	5	19	24.	4	31			
12.	1	6	25.	2	49			
13.	2	31	26.	1	9			

Table 6. SIM model coverage.

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.

Finally, while we believe that the three models we worked with are good representatives of practical models, we 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. To evaluate such a case, we executed the algorithm on a model with 10 steps out of which four incremented a single variable x. The requirements targets were in three different steps, each requiring a specific value (3, 5, and 10) for x. In this case, to cover the requirement, we need to execute the right step for covering it at the exact right time. Executing one of the steps that increments the value again when x has the correct value (3, 5 or 10) will miss the requirement for that run. Thus the probability is much lower. In such a case, we found the algorithm can find a path but reducing it to an optimal, shortest path, takes a much longer time. This is due to the experimental shortening performed by the algorithm not working so well when, even after reducing step counts, covering a requirement is still highly dependent on probabilities of step ordering.

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 6, 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.

In our experiments, the time taken to generate both the greedy and the online look-ahead optimization sets is about the same (close to 2 hours). However, with the online version this also includes the execution of the test suite, whereas with the greedy version it only includes the generation, and the execution would need to be performed separately.

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 test generation and execution 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.

The results shown in Table 5 for the single target optimizer also highlight an important finding. If one assumes a coverage requirement that would need about 40 steps to reach it, using a test length of 50 sounds reasonable. However, Table 5 shows how coverage requirements can often be more easily found by generating much longer test cases 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 6, 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 [13]. 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 assumed 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.

TEMA [15] is a MBT tool applied in the smartphone domain. It is also support online testing for Android devices, based on a form of random selection. The tool is mainly used to test applications through their UIs, similarly to the way we tested the iTrust web application through its UI in Section IV-A. TEMA targets a similar test domain in GUI-based testing but does not provide optimization approaches such as ours.

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., [16]). 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 [16] 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 [17], 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) [18] combines static analysis and observations about the running system with random inputs to guide it towards new paths. Java PathFinder (JPF) [19] 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 [20].

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 [21]. 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 [22, 23], 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 [23]. 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.

[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 Int'l. Conf. on Automated Software Engineering (ASE 2004)*, 2004.

[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. Kääramees, "Constrain-Based Heuristic On-line Test Generation from Non-Deterministic I/O EFSMs," in *7th Workshop on Model-Based Testing*, 2012.

[7] A. Groce, A. Fern, J. Pinto, T. Bauer, A. Alipour, M. Erwig and C. Lopez, "Lightweight Automated Testing with Adaptation-Based Programming," in *IEEE Int'l. Symposium on Software Reliability Engineering*, 2012.

[8] T. Kanstrén, "OSMO Tester Home Page," 2014. [Online]. Available: http://code.google.com/p/osmo. [Accessed April 2014].

[9] M. Veanes, C. Campbell, W. Schulte and N. Tillmann, "Online Testing with Model Programs," in *ESEC/FSE-13*, 2005.

[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.

[12] T. Kanstren and T. Kekkonen, "Distributed Online Test Generation for Model-Based Testing," in *Asia Pacific Software Engineering Conference*, 2013.

[13] M. Utting, "The ModelJUnit Model-Based Testing Tool," 2009. [Online]. [Accessed 17 May 2013].

[14] North Carolina State University, "iTrust: Role-Based Healthcare," 2013. [Online]. [Accessed 17 May 2013].

[15] T. Takala, M. Katara and J. Harty, "Experiences of System-Level Model-Based Testing of an Android Application," in *IEEE Int'l. Conf. on Software Testing, Verification, and Validation (ICST2011)*, 2011.

[16] P. McMinn, "Search-based testing: Past, present and future," in *3rd Int'l. Workshop on Search-Based Software Testing*, 2011.

[17] H. Baller, S. Lity, M. Lochau and I. Schaefer, "Multi-Objective Test Suite Optimization for Incremental Product Family Testing," in *IEEE Internation Conference on Software Testing, Verification and Validation (ICST2014)*, 2014.

[18] P. Godefroid, N. Klarlund and K. Sen, "DART: Directed Automated Random Testing," in *Programming Language Design and Implementation (PLDI 2005)*, 2005.

[19] C. S. Pasareanu and N. Rungta, "Symbolic Pathfinder: Symbolic Execution of Java Bytecode," in *Automated Software Engineering (ASE 2010)*, 2010.

[20] D. Saff and M. Ernst, "Reducing Wasted Development Time via Continous Testing," in *Proc. Int'l. Conf. on Software Testing and Analysis (ISSTA)*, 2003.

[21] S. Bardin, N. Kosmatov and F. Cheynier, "Efficient Leveraging of Symbolic Execution to Advanced Coverage Criteria," in *IEEE Int'l. Conf. on Software Testing, Verification and Validation*, 2014.

[22] 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.

[23] A. Arcuri, M. Z. Iqbal and L. Briand, "Random Testing: Theoretical Results and Practical Implications," *IEEE Transactions on Softw. Eng.,* vol. 38, no. 2, pp. 258-277, 2012.