Black-Box Test Generation from Inferred Models

Petros Papadopoulos and Neil Walkinshaw
Department of Computer Science
The University of Leicester, UK
Email: petros.papadopoulos@gmail.com, n.walkinshaw@le.ac.uk

***Abstract*—Automatically generating test inputs for components without source code (are 'black-box') and specification is challenging. One particularly interesting solution to this problem is to use Machine Learning algorithms to infer testable models from program executions in an iterative cycle. Although the idea has been around for over 30 years, there is little empirical information to inform the choice of suitable learning algorithms, or to show how good the resulting test sets are. This paper presents an openly available framework to facilitate experimentation in this area, and provides a proof-of-concept inference-driven testing framework, along with evidence of the efficacy of its test sets on three programs.**

I. Introduction

Automated test generation typically requires either access to source code, or to some hand-crafted model of software behaviour. Depending on the circumstances, neither of these will necessarily be available. The program in question might for example be a pre-compiled COTS component, or be an embedded system. Models are rarely available, or easily become out-dated as the program evolves.

In the absence of code or models, random (or quasi-random) testing is typically chosen as an alternative. However, this approach has several downsides. The random selection of inputs can favour the execution of 'easy-to-reach' behaviour, and miss out faulty behaviour that arises from specific input combinations. Furthermore, without access to code or runtime information, there is no basis upon which to establish how *adequate* a resulting test set is – how much confidence can be derived from a successful test set execution.

In the early eighties, Weyuker [21] and Budd and Angluin [3] examined this problem from the perspective of Machine Learning. They advanced the argument that testing and model inference are, in effect, two sides of the same coin. Testing is about selecting program inputs to elicit program behaviour that demonstrates certain properties, and model inference is concerned with deducing these properties from the observed behaviour.

With the rise of Machine Learning this idea has gradually started to gain traction. However, efforts to investigate this idea have been overwhelmingly geared towards the relatively narrow class of sequential systems that can be modelled as state machines [13], [17], [20], [19], [11] (possibly because such models are associated with an extensive range of state-machine testing techniques). Similar work on data-driven programs, of the sort that can be executed by a single command on the command-line, remains limited, and the authors are not aware of any openly available implementations (that do not require the source-code of the program under test) to facilitate this experimentation.

In this paper we present framework that is designed to support the inference-driven test generation for programs that are not sequential. Importantly, the framework is designed to be modular; it is not necessarily tied to a specific model inference or test generation framework, and can be in principle applied to any executable program, without the need for access to source code. The key contributions of this paper are as follows:

- **The Model-Inference driven Testing (MINTEST) framework.** The framework is deliberately flexible; for our proof of concept we show how to use WEKA's J48 [9] implementation of Quinlan's C4.5 algorithm [12] to infer decision trees from program executions, and use the Z3 solver [5] to generate and execute tests from it. These are analysed to produce new test inputs, and the cycle iterates.
- **An openly-available implementation.** We provide an openly-available Java implementation that can be (and is in the process of being) extended to handle different types of programs, models, and test generators.
- **An evaluation on three openly-available programs.** The evaluation shows how inference-driven testing can produce better test sets more efficiently than random testing.

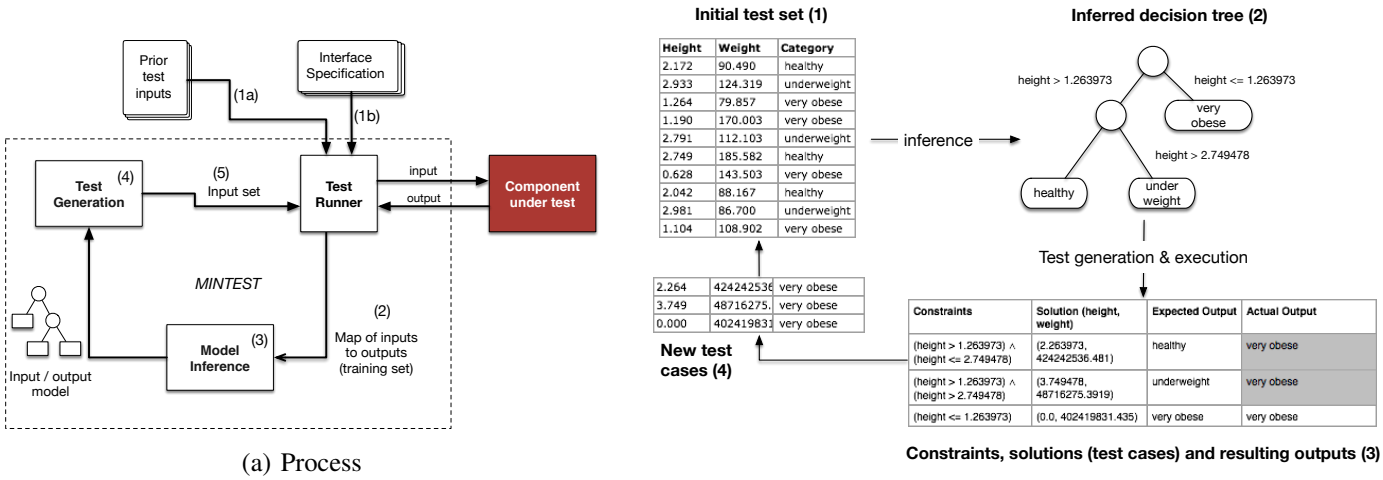
The rest of this paper is structured as follows. Section II provides the basic knowledge required to render this paper reasonably self-contained. Section III introduces our technique and implementation. Section IV contains the evaluation. Section V contains related work, and Section VI contains conclusions and future work.

II. Background

A. Inference and Testing

A general framework for the process of combining model inference with automated test generation is shown in Figure 1. This can be used to characterise most of the previous approaches. The main steps are elaborated below:

1) Any existing tests are executed and the outputs are recorded.
2) The inputs and outputs are treated as a training set, and fed to a model inference engine.
3) The resulting model provides a hypothetical approximate generalisation of the program behaviour that has been tested so far.



(a) Process

(b) BMI Example

Fig. 1. Inference-driven test generation process

4) The inferred model is used to guide the selection of new test inputs, with a view to finding input / output relations that contradict the inferred model (i.e. executing 'new' program behaviour). Previous approaches have used the inferred model to drive genetic algorithms [8], [7], or to use state machine testing strategies [13], [20], [19], [11].
5) The loop is repeated, until some termination criterion is met (e.g. no further contradictory tests can be found).

One useful 'by-product' of this approach is the inferred model. This is useful because it provides a non code-based perspective on what software behaviour has (probably) been tested. This can therefore be used to formulate novel notions of test adequacy Weyuker [21], [18].

B. Motivation

Experimental work on the interplay between testing and inference is challenging. The components involved in the test-generation cycle (inference algorithm, an accompanying model representation, test generation algorithm and test execution framework) can involve an extensive amount of implementation effort. This invariably makes it difficult to experiment with the effect of different model inference or test generation algorithms for example.

One inference-testing tool-set that has been particularly successful for sequential systems has been LearnLib [13]. One reason for this comes down to its inherent modularity, which works around the problems mentioned above. It enables the selection of different algorithms or strategies for various parts of its test-generation cycle.

The motivation for the work presented in this paper is to develop an equivalent for LearnLib, but for programs that do not necessarily operate on sequential inputs and outputs. The approach should however be modular, to enable the selection of different inference and test-generation algorithms. It should also be possible to easily direct the technique towards any program that can be executed via the command-line.

III. Approach and Implementation

Here we introduce our MINTEST framework of the loop shown in Figure 1 (a). It is deliberately modular, in that each component in the loop can be readily replaced by a variety of alternative implementations. We describe the components of our technique in terms of the components shown in Figure 1 (a). The key steps are illustrated on a small running example - a black-box BMI calculator function, shown in Figure 1 (b). The source code for the approach, an executable Jar file, and some instructions are available online[1].

a) Component Under Test: The component under test refers to any programs that can be executed with a single invocation that takes a (possibly empty) list of parameters. These parameters can either contain categorical values (e.g. String / boolean values) or numbers. The output must be a single value that is again either a categorical value or a number. Note that, given the black-box nature of the approach, there are no constraints on the implementation language or source code complexity.

For our small running example, we consider a simple routine of a BMI calculator. This takes as input someone's height (in metres) and weight (in kilograms), and returns an assessment of whether they are underweight, normal, overweight, obese, or very obese. The output illustrates what we mean by 'categorical' output. It may be encoded in reality as a string or a number, but one that can only assume a finite range of values. An example of these inputs is shown in Figure 1(b)-1.

b) Model Inference: We link up to the WEKA inference framework [9] to infer suitable models from the resulting data. Although there are numerous potential choices for algorithms, these can only be chosen if there exists a test generator that is capable of processing the inferred model. So far, we have only

[1]https://bitbucket.org/nwalkinshaw/efsminferencetool/wiki/TestGeneration

```
{
   {
          "command": "/tmp/tcas",
          "parameters":[
             {
                "name": "cur_vertical_sep",
                "type": "integer"
             },
             {
                "name": "High_Confidence",
                "type": "integer",
                "min": "0",
                "max": "1"
             },
             ...
             {
                "name": "Climb_Inhibit",
                "type": "integer",
                "min": "0",
                "max": "1"
             }
          ],
          "output-type": "string"
   }
}
```

Fig. 2. JSON interface specification for TCAS program

produced a test-generator for decision trees. So by default we currently use the C4.5 decision tree inference algorithm (more precisely, WEKA's J48 implementation of it).

Figure 1(b)-2 shows the model inferred for the given initial test set. It is read by starting from the root-node. Each path to a leaf-node represents a specific conjunction of constraints on the inputs that lead to a particular output. Given the small number of tests, this initial model is a crude oversimplification of the actual rules at play; it only considers the height of the person, and does not factor in the person's weight, for example.

c) Test Generation: The test-generation module takes an inferred model as input, and generates test cases from it. So far we have implemented, alongside a simple random test set generator, a generator for decision tree models that identifies inputs to 'cover' every path through the decision tree. This is achieved by, for each leaf-node in the tree, deriving a set of inputs that satisfy the conjunction of constraints on the branches leading to that node. We use the Z3 solver [5] to solve the constraints. If, upon execution, the program returns a different value to that provided at the leaf node, then we have found a test case that invalidates the model (which then leads to the inference of a new model and a fresh iteration of the loop).

Figure 1(b) - 3 illustrates this test-generation process. The constraints derived from the paths to the leaf nodes in the decision tree are shown in the left-most column. The solutions produced by Z3 are shown in the second column, and the expected outputs (as declared in the decision tree) are shown in the third. The final column shows the results obtained by actually running these inputs on the program. Here we see that in two cases (the shaded cells), the outputs differ from the expected outputs. In this case, these observations are added to the full test set, and the whole loop iterates.

d) Running the Tool: To commence the test-generation process, the developer has to supply a description of the interface to the component under test. In our implementation, this is structured as a simple JSON file, an example excerpt of which is shown in Figure 2. This, contains three elements: the command-line command, a list of typed parameters, and a typed output. Parameters can (in the current version) either be integers, strings, or reals.

We have included some basic capability to specify limits that are known to apply to the parameters. This enables us to guide the test-generation. Numerical values can be associated with upper and lower limits. All types can be restricted to a list of specific values. As an example, we might happen to know that the `High_Confidence` parameter in the example is a boolean that is read as an integer. Therefore we might wish to constrain any input selections for that integer to being either 1 or 0.

In addition to the JSON file, it is also possible to provide an initial set of test inputs (e.g. from an existing, albeit perhaps inadequate test set). This can provide a useful basis from which to infer the initial model. The initial test set simply takes the form of a text file where each line represents a test case, where parameter values are separated by spaces. This is the format adopted by, for example, the Software Artefact Infrastructure Repository [6].

IV. Evaluation

The research question we address in our evaluation is as follows:

> Given a basic test set that executes every outcome for a program at least once, does MINTEST improve upon its adequacy?

To investigate this, we have carried out a preliminary experiment on three programs, using mutation testing as an approximation of test adequacy (the extent to which a test set can detect faulty program behaviour). The rest of the section describes the experiment set-up and the results. However, there is also a subsection devoted to the detection of a non-mutant fault in TCAS which, though not part of the measurable experiment outcomes, is nonetheless illustrative of the ability of the technique to detect faults that might evade traditional techniques.

A. Methodology

For each program we either obtained an existing set of mutations, or generated a set with a mutation testing tool. We then selected an initial "seed" test set for the program. This was randomly generated from existing test sets (with constraints on input parameters where deemed sensible), with the sole criterion that it should contain test cases that exercise every outcome of the program at least once[2], and that these outputs must be reflected in the decision tree inferred from the set of test cases (this deliberately did not involve the source

[2]For Triangle, contain one test case for every type of triangle, for BMI contain a test case for every type of weight-category, and for TCAS, return a true or a false.

code). For Triangle and BMI this was achieved with 50 test executions. However, for TCAS, this led to an initial test set of 500.

We then generated two test sets; one with MINTEST, and the other with a random test generator. To measure the extent to which the test set improved over time, we measured the mutation score after each iteration of the MINTEST, recording the equivalent mutation score for the equivalent number of random tests. This, ultimately, provided us with data showing the respective changes in mutation scores for MINTEST and random over the course of the test generation.

B. Subject programs

Programs were selected according to two criteria. (1) they had to fit the class of system we are seeking to test (i.e. numerical / boolean / string input parameters, and a categorical output type). (2) we had to be able to subject them to mutation testing. Accordingly, they either had to have an existing set of prior mutants (as is the case with artefacts on the Software Artefact Infrastructure Repository [6]), or we had to be able to generate mutants with an existing tool such as Milu [10]. The choice of programs is listed below:

- TCAS: An air traffic collision avoidance system (implemented in C), used frequently for testing research. We used the implementation, mutations, and initial test set from the base material available on the Software Artifact Infrastructure Repository [6].
- BMI: A program that categorises patients according to their Body Mass Index, computed from their height and weight. This was previously used in previous work by the authors [7]. We selected a quasi-random initial test set (with the inputs constrained to achieve all of the possible body-mass categories at least once). We encoded the Java version as C, so that we could use Milu to generate the mutants.
- Triangle: A well known program that calculates the type of a triangle from the lengths of its edges. We used a similar strategy as above to generate an initial test set, and used Milu to generate the mutants.

C. Results and Discussion

Figure 3 shows time-series plots of the mutation scores for MINTEST and the same number of random tests. From these results we can cautiously make two observations (bearing in mind the threats to validity, to be discussed below):

Generated test sets are *at least as adequate* as equivalent random test sets. In two cases the test sets produced by MINTEST produced higher mutation scores than the equivalent number of random tests. This was especially striking with TCAS, where the mutation score for random testing stayed at 71% (30 out of 42 mutants), whereas the the MINTEST mutation score reached 83% (35 out of 42).

Generated test sets are *more efficient* than random equivalents. In all cases, the mutation scores for the MINTEST test sets increases much more rapidly than for the equivalent random test sets (if the random score does increase at all). It is striking that especially the first 10 iterations seem to yield the largest increases in mutation score. This is especially striking with the BMI example. After 6 iterations (85 tests) MINTEST reaches its maximum mutation score. This score is only eventually matched by random testing at the 40th iteration, after 767 tests.

D. Non-Mutant Failure in TCAS

Interestingly, the test sets produced for the TCAS program triggered a failure that was not the cause of a mutation. Whenever the SUT returns an unexpected output (in our case an output that does not conform to the declared output type), it is logged as an Error output. Below is an example of such an output to arise from TCAS:

```
Error output for tcas/source.alt/source.orig/tcas:
(cur_vertical_sep=1772623018 High_Confidence=1
Two_of_Three_Reports_Valid=0
Own_Tracked_Alt=601329925 Own_Tracked_Alt_Rate=0
Other_Tracked_Alt=1952921136
Alt_Layer_Value=1992045881 Up_Separation=562
Down_Separation=499 Other_RAC=0
Other_Capability=2 Climb_Inhibit=0 )
```

Executing TCAS with those inputs causes the following output:

```
$ ./tcas 1756983438 1 0 615293700 599 1069169854
108928665 1191097874 715914807 2 2 0
Segmentation fault: 11
```

Upon inspection of the source code, the root cause for the failure is that one of the inputs (`Alt_Layer_value`) is an integer value that is expected to be limited to ≤ 3, which is then used to access a value in an array of length four. In our case, we did not limit the range of `Alt_Layer_value`.

It is important to note however that it is not simply the breaching of this contract per-se that causes the segmentation fault. Setting `Alt_Layer_value` to values that are much higher than 3 (e.g. 5, 10, 150, 1500) do not cause the program to fail. It is only when set to a much higher value is chosen (in our case 1992045881) that it overshoots the target array to such an extent that a segmentation fault arises.

Of course, it can be argued that it is unsurprising that disobeying the contract of the program results in a failure. However, the Software Artifact Infrastructure Repository package for TCAS is accompanied by a set of 1608 other test cases, generated by a combination of black and white-box approaches by researchers at Siemens. Many attempt 'invalid' values for `Alt_Layer_value`, and none of them trigger a segmentation fault. In other words, here is an example of an instance where the use of inference-driven testing has highlighted a failure case that, though sought for, has not been triggered by other techniques.

E. Threats to Validity

Given the preliminary nature of this study, there are clearly several intrinsic risks to validity that need to be addressed by a further, more comprehensive study (see Section VI). The test-generation components in both MINTEST and the random approach involve a degree of randomness; by only

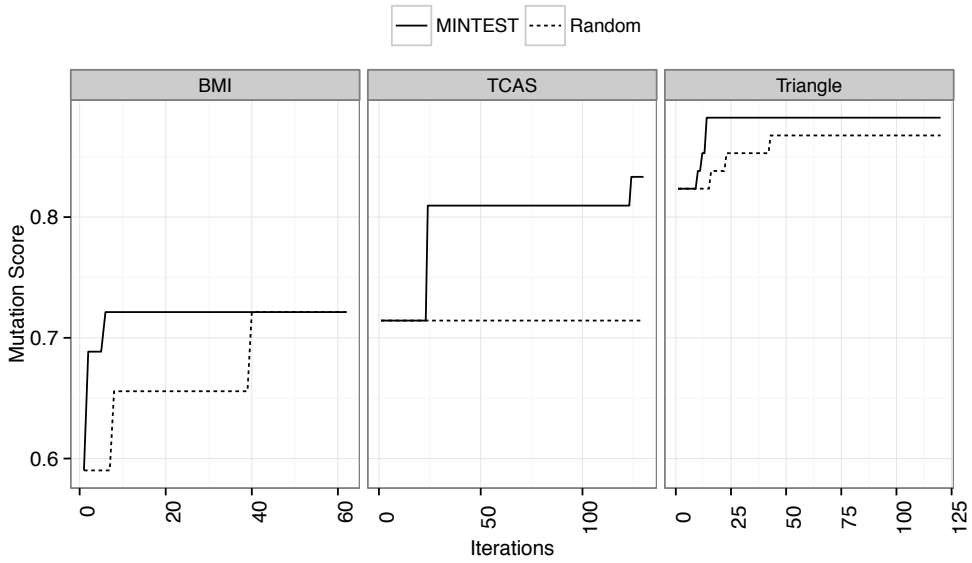


Fig. 3. Mutation Scores

providing a single run per case study there is the risk that the relative performance of the techniques is accidental as opposed to representative. Furthermore, MINTEST relies on an initial "seed" set of executions to provide an initial model. There is the possibility that the results were biased by the selection of this test set. Finally, there is the obvious threat to external validity that we have only run these experiments on three programs, and that more programs would be required to obtain a true picture of how MINTEST performs with respect to all programs of this class.

V. Related Work

There has been a substantial amount of interest in the relationship between model inference and software testing. Until the mid-nineties this work was predominantly theoretical [3], [21], [4], [15], [14], [23], [22]. This developed strong theoretical underpinnings for test adequacy in an inference context (by linking to Machine-Learning notions such as Probably Approximately Correct learning, for example [23], [18]).

Applied research into linking testing and inference has generally arisen from work that takes a state machine-centric view of software [13], [17], [20], [19], [11]. There are numerous well-established state machine inference algorithms and state machine testing algorithms, which makes the two areas particularly complementary in this respect. However, in practice, it is rarely practical to model program behaviour as a finite state machine. On the one hand, FSMs assume that the software is sequential in nature (e.g. a GUI or a network protocol), but do not apply to non-sequential functions. More importantly, they do not apply to software that operates on data; where the output is in some way contingent upon the data values of the input. This is the goal of our work.

There have been several approaches that have sought to apply (at least some parts of) the inference-testing loop to data-driven programs. The first applied approach known to the authors was by Bergadano and Gunnetti [1] in 1996, who used inference to generate test cases that are specific to a particular version of a program (assuming that other versions are available). More recently, Briand *et al.* [2] have used inference to drive the manual selection of test cases for the Category Partition method. Ghani and Clark [8] presented a technique to refine reverse-engineered Daikon models by finding test cases to contradict the models. Fraser and Walkinshaw [7] presented a model-inference driven framework that follows the same loop, and uses the same algorithms as those presented here. However, crucially, their approach is not black-box; the core test-sets are generated by conventional white-box syntax coverage.

These approaches all underpin the rationale for the MINTEST framework, in that they make a compelling case that model inference can, in parts, spur test generation. That said, none of them have been evaluated as fully-automated test generation approaches to a black-box system. It is the goal that MINTEST will form a framework that can be used in this way.

VI. Conclusions and Future Work

In this paper we have introduced our MINTEST framework. This enables the flexible implementation of the loop, first explored by Weyuker [21] and Budd and Angluin [3], of linking model inference to test generation. We provide an instantiation of this framework that shows how the WEKA implementation fo the C4.5 algorithm [12] can be linked up to Z3 [5] to generate test sets for three programs. We also show how the resulting test sets are more adequate than equivalent random test sets, and are more efficient at identifying faults.

As far as future work is concerned, in the short term, we are currently building up a stronger empirical study to address the weaknesses mentioned previously. Although the preliminary results are promising, we are keen to establish the limits of the

technique; to delineate more clearly between scenarios where the MINTEST succeeds, and where it fails.

Looking into the longer term, the flexibility afforded by MINTEST to combine different inference and test generation algorithms raises several exciting areas in which we seek to carry out further research. There are two particular questions we seek to investigate:

1) Identifying model inference algorithms that are capable of inferring accurate models with multiple-inputs (modelling programs with multiple data outputs, instead of a single categorical output).
2) Incorporating learning algorithms such as Suppport Vector Machines or Neural Nets that do not produce explicit, "testable" models, and ...
3) ... developing novel test-generation techniques for such models that are derived from the domain of Active Machine Learning [16].

REFERENCES

[1] F. Bergadano and D. Gunetti. Testing by means of inductive program learning. *ACM Transactions on Software Engineering and Methodology*, 5(2):119–145, 1996.

[2] L. Briand, Y. Labiche, Z. Bawar, and N. Spido. Using machine learning to refine category-partition test specifications and test suites. *Information and Software Technology*, 51:1551–1564, 2009.

[3] T. A. Budd and D. Angluin. Two notions of correctness and their relation to testing. *Acta Informatica*, 18(1):31–45, 1982.

[4] J. Cherniavsky and C. Smith. A recursion theoretic approach to program testing. *IEEE Transactions on Software Engineering*, 13, 1987.

[5] L. De Moura and N. Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[6] H. Do, S. G. Elbaum, and G. Rothermel. Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact. *Empirical Software Engineering: An International Journal*, 10(4):405–435, 2005.

[7] G. Fraser and N. Walkinshaw. Behaviourally adequate software testing. In *International Conference on Software Testing, Verification and Validation (ICST)*, 2012.

[8] K. Ghani and J. Clark. Strengthening inferred specifications using search based testing. In *International Conference on Software Testing Workshops (ICSTW)*. IEEE, 2008.

[9] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, and I. H. Witten. The weka data mining software: an update. *ACM SIGKDD explorations newsletter*, 11(1):10–18, 2009.

[10] Y. Jia and M. Harman. Milu: A customizable, runtime-optimized higher order mutation testing tool for the full c language. In *Practice and Research Techniques, 2008. TAIC PART'08. Testing: Academic & Industrial Conference*, pages 94–98. IEEE, 2008.

[11] K. Meinke and M. A. Sindhu. Incremental learning-based testing for reactive systems. In *Tests and Proofs*, pages 134–151. Springer, 2011.

[12] J. R. Quinlan. *C4. 5: programs for machine learning*, volume 1. Morgan kaufmann, 1993.

[13] H. Raffelt and B. Steffen. Learnlib: A library for automata learning and experimentation. In *Formal Aspects of Software Engineering (FASE)*, LNCS, pages 377–380, 2006.

[14] K. Romanik. Approximate testing and its relationship to learning. *Theoretical Computer Science*, 188(1-2):175–194, 1997.

[15] K. Romanik and J. Vitter. Using Vapnik-Chervonenkis dimension to analyze the testing complexity of program segments. *Information and Computation*, 128(2):87–108, 1996.

[16] B. Settles. *Active Learning*. Synthesis Lectures on Artificial Intelligence and Machine Learning. 2012.

[17] M. Shahbaz and R. Groz. Inferring mealy machines. In *Formal Methods (FM)*, LNCS, pages 207–222, 2009.

[18] N. Walkinshaw. Assessing test adequacy for black-box systems without specifications. In *International Conference on Testing Software and Systems, (ICTSS)*, volume 7019 of *LNCS*, pages 209–224. Springer, 2011.

[19] N. Walkinshaw, K. Bogdanov, J. Derrick, and J. Paris. Increasing functional coverage by inductive testing: A case study. In *International Conference on Testing Software and Systems (ICTSS)*, LNCS, 2010.

[20] N. Walkinshaw, J. Derrick, and Q. Guo. Iterative refinement of reverse-engineered models by model-based testing. In *Formal Methods (FM)*, LNCS, pages 305–320. Springer, 2009.

[21] E. J. Weyuker. Assessing test data adequacy through program inference. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(4):641–655, 1983.

[22] H. Zhu. A formal interpretation of software testing as inductive inference. *Software Testing, Verification and Reliability*, 6(1):3–31, 1996.

[23] H. Zhu, P. Hall, and J. May. Inductive inference and software testing. *Software Testing, Verification, and Reliability*, 2(2):69–81, 1992.