Increasing Functional Coverage by Inductive Testing: A Case Study

Neil Walkinshaw[1], Kirill Bogdanov[2], John Derrick[2], and Javier Paris[3]

[1] Department of Computer Science, The University of Leicester, Leicester, UK
[2] Department of Computer Science, The University of Sheffield, Sheffield, UK
[3] Department of Computer Science, University of A Coruña, A Coruña, Spain

Abstract. This paper addresses the challenge of generating test sets that achieve functional coverage, in the absence of a complete specification. The inductive testing technique works by probing the system behaviour with tests, and using the test results to construct an internal model of software behaviour, which is then used to generate further tests. The idea in itself is not new, but prior attempts to implement this idea have been hampered by expense and scalability, and inflexibility with respect to testing strategies. In the past, inductive testing techniques have tended to focus on the inferred models, as opposed to the suitability of the test sets that were generated in the process. This paper presents a flexible implementation of the inductive testing technique, and demonstrates its application with case-study that applies it to the Linux TCP stack implementation. The evaluation shows that the generated test sets achieve a much better coverage of the system than would be achieved by similar non-inductive techniques.

1 Introduction

The quality of a test set is conventionally measured by the extent to which it exercises the System Under Test (SUT). Various different notions of 'coverage' have been developed for different testing techniques, such as the proportion of branches / basic-blocks covered in structural source code testing, or the number of mutants killed in mutation testing (a comprehensive overview of coverage measures is provided by Zhu *et al.* [1]). Functional coverage is conventionally measured with respect to a program specification. The goal of a functional test set generator is to generate a finite set of tests that is sufficiently large and diverse to fully exercise the specification, and identify any implementation faults or failures in the process.

Unfortunately, specifications that fully encapsulate the desired system behaviour are rarely available, because they are often deemed to be too time-consuming to construct and maintain. If they exist at all, they are usually partial; for instance, it is possible to obtain a list of the main interface functions for a system, perhaps coupled with some of the more important pre-/post-conditions. However, it is unrealistic to presume the existence of a complete and up-to-date specification that could be used to ensure that the underlying system is functionally correct.

Inductive testing [2–13] offers a partial solution to this problem. It is based on the observation that testing and inductive inference are two sides of the same coin. In software testing the challenge is to identify a finite set of tests that will fully exercise some software system. In inductive inference the challenge is to work out what the (hidden) system is from a finite sample of examples of its behaviour. **The success of techniques in either area depends on the depth and breadth of the set of examples or tests**. In testing, a broader test set is more likely to lead to some unexpected / faulty state. In inductive inference, a broader set of examples will lead to a more accurate inferred model. Inductive testing exploits this symmetry between the two areas by inferring specifications from tests; the inferred model shows what has been tested already, so that the test-generator can attempt to find new, contradicting test cases. The process is iterative, and terminates once no further tests can be found that conflict with the inferred specification.

Although the idea of inductive testing is well-established (Weyuker's early work on this [2] dates back to 1983), its application has been restricted to small systems, and has not been widely adopted. This is mainly due to the fact that inductive inference techniques tend to scale poorly, coupled with the fact that some techniques are tied to very specific and expensive systematic test-generation approaches (e.g. Angluin membership queries [14]). Besides the scalability and flexibility problems, reluctance to adopt such techniques could also be due to the fact that they do not explicitly answer the following question: *To what extent does inductive testing improve on the coverage achieved by conventional testing techniques?*.

If it can be shown that inductive testing techniques can (a) be applied to large, realistic systems and (b) that they achieve better functional coverage than conventional black-box testing techniques, such techniques are bound to appeal to the broader testing community. This paper details a case-study that is intended to illustrate those two points. It uses an inductive testing framework that has been developed by the authors [13], which is flexible (allows the user to select their own testing algorithm / implementation), and can infer models from incomplete test sets with the help of heuristic inference algorithms [15, 16]. The practicality of the technique is underpinned by the fact that it incorporates the Erlang QuickCheck testing framework, which means that it can be applied to a wide range of Erlang applications and network protocols. In our case, we demonstrate its applicability by automatically constructing a test set for the Linux TCP/IP stack. The main contributions of this paper are as follows:

- The practical applicability of inductive testing is demonstrated in a practical context on a real black-box system – the Linux TCP/IP stack.
- It is shown how the system can be combined with a network-protocol testing framework to enable the testing of arbitrary network protocols.
- The *functional coverage*, and average depth of the tests is measured, and compared against the results from an equivalent non-inductive testing technique.

Section 2 discusses the background; it discusses the general problem of achieving functional test coverage, the inductive testing technique, and its problems. Section 3 presents a high-level view of the inductive testing technique. Section 4 shows how we have implemented the various aspects of the technique, and shows how this has been applied to automatically generate test sets for the Linux TCP/IP stack. It presents results that show how the coverage and depth of test sets generated by the inductive testing technique compare favourably against test sets generated by an equivalent non-inductive version of the technique. Finally, section 5 concludes, and discusses future work.

2 Background

2.1 Context: Achieving Functional Test Coverage without a Model

This paper is concerned with the challenge of producing a test set for a system that is a black-box, and for which there is at best only a partial specification. Exhaustively enumerating every input is infeasible for most realistic programs, and the conventional approach of using random inputs is only likely to exercise those states of the system that are easy to reach by chance. Such approaches often fail to reach those areas of the state-space that are most likely to elicit unexpected program behaviour, such as an unhandled exception or a straightforward program failure.

As opposed to conventional model-based testing, the aim is not to demonstrate that the behaviour of the system is functionally correct. This is impossible without a complete and reliable specification. Instead the aim is to identify a set of test cases that fully exercise the SUT, in the hope that one of these will elicit program behaviour that is obviously incorrect, such as a program failure. Conventional coverage-driven testing approaches aim to fully exercise the SUT in terms of its source code or specification [1]. In the absence of either of those, the aim in our case is to obtain a test set that fully exercises SUT in terms of its observable behaviour.

The definition of "observable behaviour" depends on the characteristics of the SUT. The observable behaviour of a network protocol would be the sequences of outputs that are returned in response to sequences of inputs. The observable behaviour of a continuous function would be the numerical value that is returned for a particular input value. The ultimate aim is to explore the full range of functionality offered by the SUT, where this functionality is manifested in the variation of outputs in response to different inputs.

2.2 Inductive Testing

A test set is deemed to be *adequate* if it achieves a given level of coverage. Many coverage measures exist to suit different testing techniques [1], such as source code branch coverage, or transition coverage in state machines, etc. In our setting we cannot count on access to a specification or on access to the source code.

One solution to the above problem is offered by a technique that was first outlined by Weyuker [2] almost 30 years ago. She related the challenge of identifying an adequate test set to the machine-learning field of inductive inference, where the challenge is to infer a model from a finite set of examples. She observed that the two problems are in effect two sides of the same coin. An accurate model can only be inferred with a sufficiently broad set of examples. Importantly from a testing perspective, a better model will ultimately result in a more comprehensive test set.

Weyuker suggested that this intuitive symmetry could be exploited, and it is this idea that forms the basis for what is referred to here as inductive testing. The idea is as follows: a set of tests can be considered *adequate* if, when fed to an inductive inference engine, the resulting model is equivalent to the SUT[4]. Since then, the idea of combining inductive inference with testing has reappeared in various guises [3–10, 17, 11–13].

Problem: flexibility and scale Current inductive testing approaches are hampered by the fact they are often tied to very specific systematic testing strategies, which in turn lead to problems in terms of scalability. Testing strategies are designed to produce a test set that will, when used as a basis for inference, produce a model that is *exact*. However, obtaining a sufficiently large test set to do so is often infeasible in practice, especially when tests are expensive to execute. Though cheaper, heuristic approaches have been proposed [13], their applicability has not been demonstrated with respect to a realistic black-box system. None of the proposed techniques have been explicitly evaluated in terms of their ability to achieve functional coverage of substantial black-box systems.

3 A Flexible Inductive Testing Technique

This section describes an inductive testing technique that is designed to be flexible. It permits the developer to choose a model-based testing technique to suit the circumstances (i.e. the complexity of the software system). To account for incomplete test sets, it adopts heuristics to infer models. Every time a model is inferred, the selected testing strategy is invoked to generate a test set from the model, which is executed on the SUT. This process continues until a set of tests has been generated that infer a model for which no further conflicting tests can be generated.

For this paper, we will restrict discussion to systems that can be modelled and tested as deterministic Labelled Transition Systems (LTS's).

Definition 1 (Labelled Transition System (LTS)) *A LTS is a quadruple* (Q, Σ, Δ, q_0)*, where* Q *is a finite set of states,* Σ *is a finite alphabet,* $\Delta : Q \times \Sigma \rightarrow$

[4] Although this is the essential idea, she considered a different problem-setting to the one considered here; her setting presumed that the system was white-box, that there was also a specification, and that the inductive inference engine was inferring executable source code as opposed to arbitrary models.

```
Input: $Prog$, $Alphabet$
Data: $Alphabet, Pos, Neg, test, fail, failedLTS$
Uses: $inferLTS(T^+, T^-), generateTests(LTS)$
Uses: $runTest(t, Prog), generateInit(Alphabet)$
Result: $LTS$
1  $Pos \leftarrow \emptyset$;
2  $Neg \leftarrow \emptyset$;
3  $LTS \leftarrow generateInitLTS(Alphabet)$;
4  $Test \leftarrow generateTests(LTS)$;
5  foreach $test \in Test$ do
6      $(trace, pass) \leftarrow runTest(test, Prog)$;
7      if $pass$ then
8          $Pos \leftarrow Pos \cup \{trace\}$;
9          if $trace \in \Sigma^* \setminus L(LTS)$ then
10             $LTS \leftarrow inferLTS(Pos, Neg)$;
11             $Test \leftarrow generateTests(LTS)$;
12
13     else
14         $Neg \leftarrow Neg \cup \{trace\}$;
15         if $trace \in L(LTS)$ then
16             $LTS \leftarrow inferLTS(Pos, Neg)$;
17             $Test \leftarrow generateTests(LTS)$;
18
19     end
20 end
21 return $LTS$
```

Algorithm 1: Basic iterative algorithm

Q is a partial function and $q_0 \in Q$. This can be visualised as a directed graph, where states are the nodes, and transitions are the edges between them, labelled by their respective alphabet elements.

Some parts of this section will refer to the *language* of the LTS. This is defined as follows.

Definition 2 (The Language of an LTS) *For a state p and a string w, the extended transition function $\hat{\delta}$ returns the state p that is reached when starting in state p and processing sequence w [18]. For the base case $\hat{\delta}(q, \epsilon) = q$. For the inductive case, let w be of the form xa, where a is the last element, and x is the prefix. Then $\hat{\delta}(q, w) = \delta(\hat{\delta}(q, x), a)$.*

Given some LTS A, for a given state $q \in Q$, $L(A, q)$ is the language of A in state q, and can be defined as: $L(A, q) = \{w | \hat{\delta}$ is defined for $(q, w)\}$. The language of a LTS A can be defined as: $L(A) = \{w | \hat{\delta}$ is defined for $(q_0, w)\}$.

The inductive testing process is presented in algorithm 1. It takes the alphabet of the machine as input (i.e. the message types that can label transitions), and uses these to generate its initial most general machine (line 3). This is then fed to a model-based tester to generate test cases (line 4). For each test case, if it has passed, its trace $trace$ is added to the set of traces that are positive Pos and if it fails it is added to the set of negative traces Neg. In either case, the test case is checked against the current LTS model to make sure that it produced the correct result. If this isn't the case, a conflicting test case has been discovered, and a new LTS is inferred to account for the new traces (lines 10 and 16). When this is the case, a new set of test cases $Test$ is generated, and the whole process iterates. The rest of this section provides a more in-depth description of the key components in algorithm 1. The *generateInitLTS* function is described below, followed by a more detailed description of the processes used to test and infer the models.

To begin with the inductive testing process an initial model is needed. In the algorithm this is generated by the *generateInitLTS* function on line 3. No prior knowledge is required about the (hidden) state transition structure of the subject system, but it does assume that the set of labels (or possible software inputs) is known. An initial LTS is a simple transition system that is produced where any sequence in Σ^* is valid. This will always consist of a single state, with one looping transition that is labelled by all of the elements in Σ. Formally, $Q = \{q_0\}$, $\forall \sigma \in \Sigma$, $\delta(q_0, \sigma) = q_0$. The tests that are generated from this will can be used by the inference process to produce more refined models.

3.1 Test Generation and Execution

The *generateTests* function represents a conventional model-based test set generator. It takes as input a model in the form of a LTS and generates a set of tests (paths through the state machine) that are expected to be either be possible or impossible in the implementation.

Given a LTS A and a test case t, the execution of a test case by the *runTest* function results in a tuple $(test, Pass)$, where $test$ is the test (if it failed, its last element is the point at which it failed), and $Pass$ states whether it passed or failed. The algorithm then matches this against the expected outcome; if $Pass == false$ but $t \in L(A)$, or $Pass == true$ but $t \notin L(A)$, there is a conflict and A has to be re-inferred.

The *generateTests* function can invoke a range of model-based testing algorithms. One the one hand there are 'formal' testing techniques that aim to guarantee certain levels of coverage of the given state machine. On the other hand, it is possible to adopt less rigorous but cheaper alternatives.

3.2 Model Inference

The inference process is denoted by the $inferLTS$ function. Its purpose is to infer an LTS from a set of test cases. The Evidence Driven State-Merging (EDSM) method [15] is currently accepted to be the leading approach for inferring LTS's from sparse sets of examples.

Conceptually, given a collection of test executions, one would expect that the executions traverse the same states in the SUT multiple times. State-merging

Pos:
$< Listen/-, Syn/Syn + Ack, Ack/-, Close/Fin >$
$< Listen/-, Syn/Syn + Ack, Ack/-, Fin/Ack >$
Neg:
$< Listen/-, Syn + Ack/Ack >$
$< Listen/-, Syn/Syn + Ack, Ack/-, Ack/- >$
$< Connect/SynAck/- >$
$< Ack/- >$

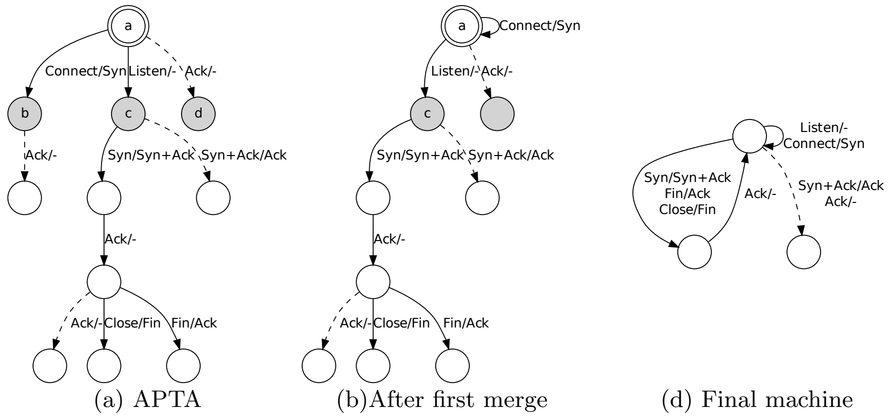


Fig. 1. Augmented Prefix Tree Acceptor and illustration of merging. Dashed transitions are deemed to be invalid, multiple transitions between states are drawn as single transitions with multiple labels

approaches arrange the presented set of traces into one large tree-structured state machine that exactly represents the executions, and then proceed to merge those states that are deemed to correspond to the same SUT state, with the aim of producing the minimal, most general LTS. The benefit of EDSM approach versus other traditional approaches is that it uses heuristics to choose suitable state-pairs, weighing up the likelihood that two states are equivalent in terms of their outgoing paths of transitions (i.e. their future behaviour). This has been shown to substantially increase the accuracy of the final machine [15, 16].

A brief illustration will be provided with respect to a small part of the TCP example from the case study. Let us assume that we have reached a point where six tests have been executed, resulting in the traces Pos and Neg shown in figure 1. These can be aggregated into a single tree - referred to as an *augmented prefix tree acceptor (APTA)* [15], shown in (a). The tree is constructed so that any identical execution-prefixes lead to the same state, and any unique suffixes form unique branches in the tree. The APTA represents the most specific and precise LTS possible, and exactly corresponds to the provided sets of executions, not accepting any other sequences.

The goal of the inference is to identify states in this tree that are actually equivalent, and to merge them. The merging process takes two states q and q'.

In effect, the state q' is removed, all of its incoming transitions are routed to q instead and all of its outgoing transitions are routed from q. Every time a pair of states is merged, the resulting state machine may be non-deterministic (new transitions from q may carry the same labels as old transitions). Non-determinism is eliminated by recursively merging targets of non-deterministic transitions. For a more detailed description, the reader is referred to previous work by Dupont *et al.*[16].

The merging process is iterative - many subsequent merges are required to reach the final machine. At each iteration, a set of state-pairs is selected using the Blue-Fringe algorithm [15] (an effective search-windowing strategy that restricts the set of state pairs to be evaluated at each iteration to a subset that are most likely to be suitable merge-candidates). Each candidate pair is assigned a heuristic score, which indicates the likelihood that the states are equivalent. The score is computed by comparing the extent to which the suffixes of each state overlap with each other[5]. A pair of states is incompatible if a sequence is possible from one state, but impossible from the other - this leads to a score of -1. Any pairs with non-negative scores can potentially be merged. Once the scores have been computed, the pair with the highest score is merged, and the entire process of score-based ranking and merging starts afresh, until no further pairs can be merged.

To provide an intuition of the scoring process, we refer back to the example prefix-tree in Figure 1 (a). State 'a' is marked red (a double-circle in the figure), and the states 'b', 'c' and 'd' are marked blue (shaded in the figure - this is the 'blue fringe'). The EDSM algorithm considers any red-blue pair with a non-negative score as a possible merge-candidate. Pairs (a,c) and (a,d) have a score of 0, because they share no overlapping outgoing label-sequences. Pair (a,b) has a score of 1 (the event $Ack/-$ is impossible from both states) and, since this pair produces the highest score, it is selected to be merged. The result is shown in Figure 1 (b). This process continues until no further valid merges can be identified, and the final machine is shown in (c).

4 Testing the Linux TCP Stack

To illustrate the inductive testing approach, and to demonstrate its ability to improve functional coverage, we use it to explore the behaviour of the Linux TCP stack [19, 20]. For this study we assume a certain, limited degree of prior knowledge about the system. We know the set of messages that can be sent to the stack, and how they affect its data-state (these are specified in the TCP RTP documents [19]). For the sake of assessing our inductive testing technique, we will assume that there is no prior knowledge of the order in which these messages can be sent. We also assume that we have no access to the internals of the stack itself, but we can reliably reset it (by restarting the Linux networking service).

[5] The Blue-Fringe algorithm ensures that the suffixes of one state are guaranteed to form a tree (i.e. a graph without loops), which facilitates this score computation.

The challenge for us is to generate a set of test cases that extensively exercises the functionality of the stack, by eliciting a range of behaviours that is as diverse as possible. This section will show how the induction testing approach is able to elicit a much broader range of behaviour than the standard alternative of attempting random input combinations.

4.1 Inductive Testing Infrastructure

The case study exactly follows algorithm 1. Here we show how the various algorithm functions have been implemented. Specifically we describe the implementation of the three key functions: *generateTests*, *runTest*, and $inferLTS$. The implementation elaborates on earlier work by the authors [13], and can in principle be applied to test any system where the behaviour is based on an LTS (if this is not the case, the $inferLTS$ and *generateTests* functions have to be adapted accordingly).

This case study involves running tests by sending messages over a network. Due to its comprehensive networking infrastructure, we use the Erlang programming language and its Open Telecom Platform (OTP) libraries [21]. This is exploited by the network-testing interface developed by Paris and Arts [20] (described in detail below). This distributed mechanism can be coupled with the powerful QuickCheck model-based testing infrastructure [22] (also described below), to provide a comprehensive network protocol testing framework.

Generating tests with QuickCheck (the *generateTests* function) For the model-based test set generation, we use the QuickCheck framework [22], a tool that has proved popular for the model-based testing of Erlang and Haskell programs. It has become one of the standard testing tools used by Erlang developers. The model is conventionally provided by a developer, either as a set of simple temporal logic properties, or as an abstract state machine.

We use abstract state machine models in this case study. Abstract state machine models are constructed by taking a labelled transition system (see definition 1). Each transition function in Δ is then associated with a transformation on a data state M, along with a set of optional pre-/post-conditions that have to hold when the function is executed.

QuickCheck generates tests with the help of *generator* functions. These are functions that, for a given data type, will produce a suitable random input. QuickCheck has several built-in generators, which can be used to produce random inputs for simple data types, such as integers, or lists of integers. To illustrate the use of these generators, a very simple property is provided below. The `prop_reverse` property takes a list of integers as input. It tests the `lists:reverse` function by ensuring that, when applied twice to a list, it returns the original list. The call to `list(int())` generates a list of random length, populated with random integers. For a more extensive example, including an abstract state machine example, the reader is referred to Walkinshaw *et al.*[13].

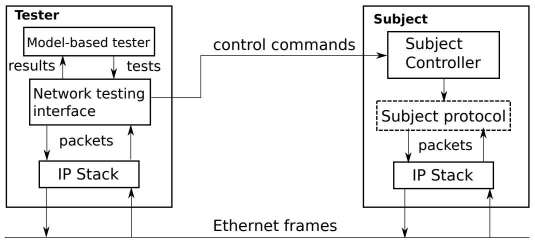


Fig. 2. Communication between tester and subject

```
prop_reverse() ->
      ?FORALL(Xs,list(int()),
      lists:reverse(lists:reverse(Xs)) == Xs).
```

The network-testing interface (the *runTest* function) Paris and Arts [20] developed a network protocol testing framework that enables model-based testing of network protocols on remote machines. The basic process is shown in figure 2. Two channels are used to interact with the system under test. One channel is the network itself, messages are synthesised by the tester and sent to the subject, and responses from the subject are sent back, which are monitored by the tester (using a network sniffer). The other channel is an Erlang communication channel that is linked to an Erlang process that is running on the SUT. It is used to control the SUT when necessary. For example if we want to test the TCP protocol in a scenario where the SUT initiates communication, this can be triggered by sending a command via this channel to the controller on the SUT, which will in turn make the stack send the corresponding message across the network link to the tester.

The network-testing interface is implemented on top of QuickCheck. The abstract state machine model keeps track of the necessary data elements (i.e. the last message received from the SUT, the Port number and IP address). For each possible message that can be sent to the SUT, a QuickCheck generator function is used to construct and send the suitable TCP packet and, if appropriate, to wait for a response from the SUT and update the data state in the abstract state machine. For a full description, the reader is referred to Paris and Arts [20].

For the sake of extensively testing TCP stacks with this framework, Paris and Arts produced a comprehensive QuickCheck model of the stack. For the inductive testing process discussed in this paper, we use a skeleton version of this model, having removed the LTS. This is instead replaced with the "universal" LTS, where every function is always possible (as generated by $generateInitLTS$ – see definition in section 3).

LTS Inference (the *inferLTS* function) Walkinshaw *et al.* have developed the openly-available StateChum model inference framework [23, 13] that was used to infer the models. It implements the Blue-Fringe state merging approach that is presented in section 3.2. A front-end was developed that, given a template QuickCheck model, would append the appropriate inferred labelled transition system, and so produce a suitable model that could be used to generate test sets for the following iteration. Usually, the output LTS generated by StateChum is displayed on-screen. To enable its use in an inductive testing context, an extension for StateChum has been developed that converts the output LTS into a QuickCheck model.

4.2 Exploring the TCP Stack Behaviour by Inductive Testing

The three functions (*generateTests*, *runTests* and *inferLTS*) are linked together by a simple Unix Bash script. Test sets are executed until a test fails. When this is the case, a new model is inferred, and the testing process starts afresh. The process will either terminate when no more conflicting tests can be found or by simply putting a limit on the number of test runs.

The basic rationale for inductive testing is that the ability to infer a model provides a picture of what has already been tested, enabling the test set generator to generate more elaborate, diverse test sets the next time round. To assess whether this is the case we compare the test sets that are generated by inductive testing to those that are generated by an identical set-up[6], but without the ability to replace the model with inferred models (i.e. where the model is always stuck on the most general model produced by the *generateInitLTS* function).

For both runs, the limit was set to 285 iterations (i.e. 285 test sets could be generated and executed). This limit was chosen because it corresponded to a couple of hours of test runs (allowing for time-outs when unexpected packet-sequences resulted in deadlocks). In practice, it would be up to the developer to select a suitable limit, based on the constraints of the project.

Measuring coverage and depth Every time a test is executed, the sequence is recorded to a text file, and it is prefixed by a "+" if it passed, or a "-" if it failed (a failure leads to the termination of the current test set execution and the generation of a new one). At any given point we are interested in the complete set of tests, i.e. all of the tests that were generated in all previous iterations leading up to that point.

The suitability of a test set is measured by the extent to which it covers the functionality of the SUT, coupled with its ability to reach "deep" states – states that require an elaborate combination of inputs to reach, and are hard to reach by random inputs. The metrics that are used to assess these are presented below:

Functional coverage There is no accepted means of measuring functional coverage without access to an existing specification. The conventional way to estimate it so far has been to measure code-coverage, however this merely provides a crude

[6] There are several systematic black-box testing techniques that would outperform this naïve approach, but we choose this one for the sake of control, to ensure that the only factor to make a difference in the assessment is the ability to infer models.

approximation of the extent to which the actual functional behaviour of the program has been covered. Indeed, it is the recognition of this fact [2] that spawned the idea of inductive testing in the first place. The model that we infer from the test sets provides a functional perspective on the test sets, and in this work we use the inferred model as a basis for measuring the functional coverage. A larger model means that the test set exercises a broader range of SUT behaviour, because it is able to distinguish between lots of different states (otherwise the inference approach would simply merge them together). So, to assess functional coverage we count the number of individual transitions in the model – these can be of two kinds; conventional state transitions, and state transitions that lead to a failure / termination state.

Although this measure seems to be more appropriate than code-coverage, it should still only be interpreted as an indicator. The EDSM inference technique operates on heuristics, and is therefore open to error. The addition of a test to a test set could under certain circumstances produce a machine that is slightly smaller than the previous version. Nonetheless, as will be shown in the results, such dips tend to be very small, and the measure still presents a good overview of the breadth of the test set.

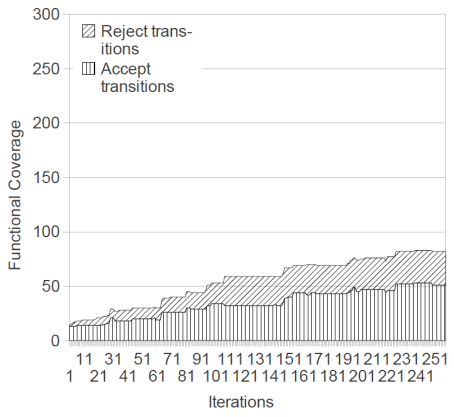
Test depth It is generally acknowledged that longer test sequences tend to lead to a higher level of coverage [24]. A long sequence of steps can be required to put a system into a configuration where it is possible to reach particular parts of the system that would remain unreachable otherwise. Every time a test set is executed in this case study, the average length of its tests is recorded. Longer test sequences imply that a test set is reaching states in the system that are harder to reach.

4.3 Results

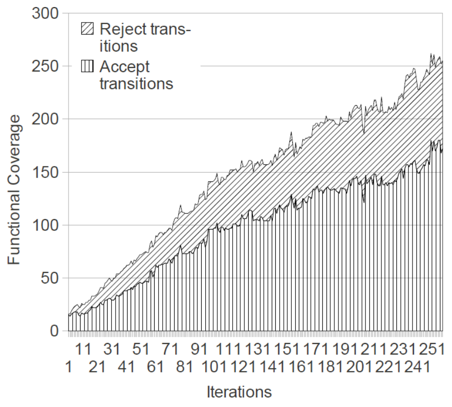
Figures 3 (a) and (b) compares the functional coverage achieved by inductive testing, and compares it to the level of coverage that is achieved by naïve non-inductive testing. The coverage is plotted for every iteration. This is split between transitions that lead to a failing / terminating state, and transitions that lead to a normal state.

The charts show that the inductive testing technique is better at exploring new system behaviour. It finds a much larger number of unique tests, and does so at a much faster rate. The non-inductive approach never surpasses 83 transitions, whereas this level is achieved within the first 60 iterations by the inductive testing approach. The charts show clearly that, by relying on random tests alone, it is much harder to identify a diverse test set that thoroughly exercises the SUT. The coverage of the system only increases very slowly as the process iterates.

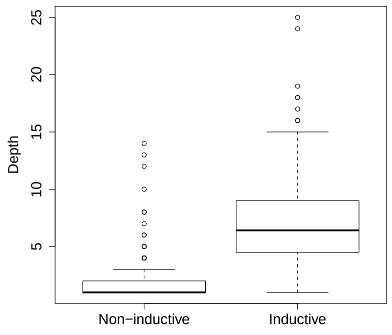
The difference between the test sets generated by the two approaches is also illustrated in figure 4. This figure shows the APTA's that are generated from the final test sets. Figure (a) is generated by the non-inductive version; although it represents a similar number of tests, many of them are simple repetitions, and do not lead to unexplored behaviour, meaning that the APTA is much smaller.



(a) Non-inductive testing



(b) Inductive testing



(c) Box-plot that compares average depths of test cases

Fig. 3. Functional coverage and comparison of average test case depth

Figure (b) on the other hand is much larger; there are far fewer repetitions amongst the tests because the inferred model encourages the discovery of new test sequences that build on the outcomes of previous tests.

Inductive testing leads to the generation of longer test sequences, which has been shown to be a significant factor in the ability to achieve high levels of test coverage [24]. Without the ability to build on the knowledge gained through previous test cases, the non-inductive approach relies on randomly finding sequences to find "deep" states – states that can only be reached by a particular test sequence. In inductive testing, these sequences are remembered; they form part of the inferred model, making it easy for subsequent tests to revisit and explore these deep states. Consequently, there is a substantial difference in the average test sequence length, as shown in the box plots in figure 3(c). The limits of the box denote the upper and lower quartiles, the line through the middle

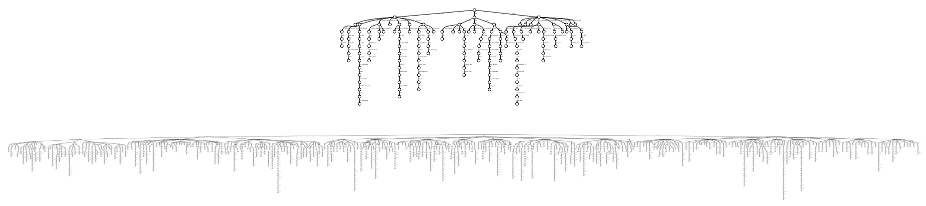


Fig. 4. Visual comparison of the APTAs generated for the non-inductive tests (on top) and the inductive tests (below)

represents the median, the whiskers mark the maximum and minimum values, and the small circles denote outliers.

Summary The use of test outputs to infer a model of system behaviour is a valuable tool for the further exploration of system behaviour, and is practical. This is demonstrated in the results above. With no prior knowledge about the possible sequences in which messages can occur, the inductive testing approach has nonetheless managed to build an extensive and diverse set of test cases that exercise the TCP stack. In comparison to the non-inductive approach, the set of test cases is much more diverse, and also manages to consistently reach states at a greater depth.

5 Conclusions and Future Work

The main aims of this paper were to show that inductive testing can be applied to realistic black-box systems, and to demonstrate that inductive testing can achieve a better functional coverage of the system than conventional non-inductive strategies. The presented inductive testing technique is flexible; it is not necessarily tied to a specific inference or testing technique, but a framework is provided that enables the two to interact, and to feed off each other. For our case study, we selected a realistic system; the Linux TCP/IP stack. We used an heuristic inductive inference technique that has been shown to deal well with incomplete test sets [15, 16, 23], and we combined this with a simple black-box network testing framework [20] that is built on top of the well-established QuickCheck Erlang testing framework [22].

The approach is flexible; different combinations of inference and testing techniques could produce better results. However, the main purpose was to demonstrate that inductive testing is a practical technique, and can produce better, more efficient test sets than naïve black-box alternatives. The best combinations of techniques will invariably depend on the characteristics of the SUT, and is one of the main areas we intend to investigate in future work.

The test set generation technique that was used to generate the test cases (both for the non-inductive and inductive cases) was deliberately simple, to ensure that any improvements in the results were not an artefact of the test-generation technique, but were entirely due to the inferred model. In practice, it makes sense to combine the model induction with a more sophisticated test-generation procedure. The QuickCheck framework has several facilities that can enable this. For example, it is possible to associate probabilities with different transitions in the LTS, to ensure that certain areas of the machine are tested more often than others. Future work will look into exploiting such features, to emphasise the exploration of new areas in the SUT that have not been explored by previous tests.

In our case study, the number of iterations were restricted to 285. This sufficed to serve its purpose of showing that inductive testing produces better test sets than non-inductive testing. However, it does raise the question of when the process should be terminated in general; when is the test set good enough? This is the question that originally motivated Weyuker to devise the inductive testing process [2]. She envisaged that test generation should continue until no tests could be found that conflict with the inferred model (or program in her case). In practice, the complexity of the SUT and the selected testing technique may make it unrealistic to achieve this goal. The convention is to declare a time-limit, and to simply execute as many tests as possible [17]. Even if this approach is adopted, this work has shown that the inferred model can be used to provide an estimation of functional coverage during the testing process, and there is a strong argument to be made for the fact that this is more suitable than the convention of using code-coverage as an approximation.

Acknowledgements Walkinshaw and Bogdanov are supported by the EPSRC REGI (EP/F065825/1) and STAMINA (EP/H002456/1) grants. Derrick and Walkinshaw are supported by the EU FP7 ProTest project.

References

1. Zhu, H., Hall, P.A.V., May, J.H.R.: Software unit test coverage and adequacy. ACM Computing Surveys **29**(4) (December 1997) 366–427
2. Weyuker, E.: Assessing test data adequacy through program inference. ACM Transactions on Programming Languages and Systems **5**(4) (October 1983) 641–655
3. Bergadano, F., Gunetti, D.: Testing by means of inductive program learning. ACM Transactions on Software Engineering and Methodology **5**(2) (1996) 119–145
4. Zhu, H., Hall, P., May, J.: Inductive inference and software testing. Software Testing, Verification, and Reliability **2**(2) (1992) 69–81
5. Zhu, H.: A formal interpretation of software testing as inductive inference. Software Testing, Verification and Reliability **6**(1) (1996) 3–31
6. Harder, M., Mellen, J., Ernst, M.: Improving test suites via operational abstraction. In: Proceedings of the International Conference on Software Engineering (ICSE'03). (2003) 60–71
7. Xie, T., Notkin, D.: Mutually enhancing test generation and specification inference. In: Proceedings of FATES 2003. Volume 2931., Springer (2003) 60–69

8. Berg, T., Grinchtein, O., Jonsson, B., Leucker, M., Raffelt, H., Steffen, B.: On the correspondence between conformance testing and regular inference. In: Fundamental Approaches to Software Engineering (FASE'05). Volume 3442 of LNCS., Springer (2005) 175–189
9. Raffelt, H., Steffen, B.: Learnlib: A library for automata learning and experimentation. In Baresi, L., Heckel, R., eds.: FASE. Volume 3922 of Lecture Notes in Computer Science., Springer (2006) 377–380
10. Bollig, B., Katoen, J., Kern, C., Leucker, M.: Smyle: A tool for synthesizing distributed models from scenarios by learning. In: Proceedings of Concurrency Theory,CONCUR 2008. Volume 5201 of Lecture Notes in Computer Science., Springer (2008) 162–166
11. Shahbaz, M., Groz, R.: Inferring mealy machines. In: Proceedings of Formal Methods (FM'09). Volume 5850 of Lecture Notes in Computer Science., Springer (2009) 207–222
12. Raffelt, H., Merten, M., Steffen, B., Margaria, T.: Dynamic testing via automata learning. STTT **11**(4) (2009) 307–324
13. Walkinshaw, N., Derrick, J., Guo, Q.: Iterative refinement of reverse-engineered models by model-based testing. In: Proceedings of Formal Methods (FM'09). Volume 5850 of LNCS., Springer (2009) 305–320
14. Angluin, D.: Learning Regular Sets from Queries and Counterexamples. Information and Computation **75** (1987) 87–106
15. Lang, K., Pearlmutter, B., Price, R.: Results of the Abbadingo One DFA Learning Competition and a New Evidence-Driven State Merging Algorithm. In: Proceedings of the International Colloquium on Grammar Inference (ICGI). Volume 1433. (1998) 1–12
16. Dupont, P., Lambeau, B., Damas, C., van Lamsweerde, A.: The QSM Algorithm and its Application to Software Behavior Model Induction. Applied Artificial Intelligence **22** (2008) 77–115
17. Pacheco, C., Lahiri, S., Ernst, M., Ball, T.: Feedback-directed random test generation. In: Proceedings of the International Conference on Software Engineering (ICSE'07), IEEE Computer Society (2007) 75–84
18. Hopcroft, J., Motwani, R., Ullman, J.: Introduction to Automata Theory, Languages, and Computation, Second Edition. Addison-Wesley (2001)
19. Postel, J.: Transmission control protocol. Technical Report 793, DDN Network Information Center, SRI International (September 1981) RFC.
20. Paris, J., Arts, T.: Automatic testing of tcp/ip implementations using quickcheck. In: Erlang '09: Proceedings of the 8th ACM SIGPLAN workshop on Erlang, ACM (2009) 83–92
21. Armstrong, J.: Programming Erlang: Software for a Concurrent World. Pragmatic Bookshelf (July 2007)
22. Claessen, K., Hughes, J.: Quickcheck: A Lightweight Tool for Random Testing of Haskell Programs. In: Proceedings of the International Conference on Functional Programming (ICFP). (2000) 268–279
23. Walkinshaw, N., Bogdanov, K., Holcombe, M., Salahuddin, S.: Reverse Engineering State Machines by Interactive Grammar Inference. In: 14th IEEE International Working Conference on Reverse Engineering (WCRE). (2007)
24. Arcuri, A.: Longer is better: On the role of test sequence length in software testing. In: Proceedings of the International Conference on Software Testing, Verification and Validation (ICST'10). (2010)