Minimizing the Number of Inputs while Applying Adaptive Test Cases

Guy-Vincent Jourdan, Hasan Ural and Nejib Zaguia
School of Information Technology and Engineering
University of Ottawa,
800 King Edward Avenue
Ottawa, Ontario, K1N 6N5, Canada
{gvj, ural, zaguia}@site.uottawa.ca

Keywords:
Software engineering, adaptive testing, adaptive test cases, minimization of test inputs

1. Introduction

State-based formalisms such as Finite State Machine and its derivatives have been used extensively for the specification of the externally observable behavior of a wide range of reactive systems [3-7]. A use of such specifications is to construct a set of test cases to be employed during testing of potential implementations of the specified system. Test cases constructed from such a specification are in the form of sequences of pairs of test input and the corresponding expected output (as in preset testing), unless it is recognized that there are more than one valid expected response and thus the next test input depends on the actual output produced in response to the current input (as in adaptive testing). If the latter is the case, then the test case is in the form of a tree, called adaptive test case. For example, Figure 1 depicts an adaptive test case where the test inputs are given in the nodes, the expected outputs are given on the edges, and the verdicts are given in the leaf nodes. A verdict is the representation of the test result which is either pass, fail, or inconclusive. A test language, called Testing and Test Control Notation (TTCN) [1] formalizes this notion and has been used for the description of adaptive test cases for testing internet related systems, services and protocols, mobile communication systems, middleware platforms, object- and component-based systems, Web services and embedded systems where there may be a variety of expected responses or options that are left to the discretion of the implementers.

During the application of a set of adaptive test cases to an implementation under test (IUT), the IUT is reset to its initial state after the application of each adaptive test case. A major component of the cost of applying a set of adaptive test cases is given in terms of the number of inputs that are applied during testing. For a deterministic IUT, it is reasonable to assume that it is unnecessary to execute on the IUT twice with the same sequence of inputs and that an execution of an IUT on a sequence of inputs subsumes the execution of the IUT on any prefix sub-sequence. Under this assumptions, it is sometimes possible to deduce the IUT's response to an adaptive test case t_2 from the response of another adaptive test case t_1 that has already been applied. When this is the case, there is no need to apply t_2 which leads to the reduction of the cost of testing.

Hence, it might seem that ordering the set of adaptive test cases might yield a substantial reduction in the number of test inputs during the application of these test cases. However, it is shown that the problem of finding an optimal order of application of a set of adaptive test cases is NP-hard [2]. In order to elude the problem of finding an optimal order, this paper proposes the simultaneous application of subsets of a given set of adaptive test cases to a given IUT, proves a lower bound on the number of inputs that need to be sent to the IUT for the application of the set of adaptive test cases and then presents an algorithm for the application of the set of adaptive test cases to the IUT that achives this lower bound.

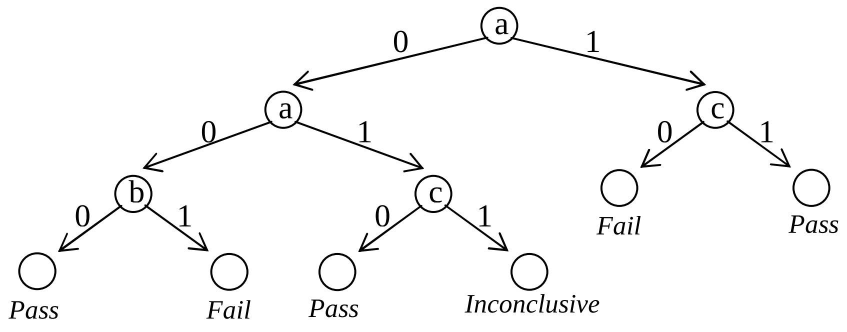

Figure 1. An adaptive test case

2. Optimizing the Number of Test Inputs

Let T={t_1, t_2, …, t_n} be a set of adaptive test cases and IUT be a given deterministic implementation. The sequence of inputs that is sent during the application of an adaptive test case (henceforth called test) t to an IUT is called the IUT's *tested word* of t and written $L_{t,IUT}$. The set of all tested words of a test set T forms a language, which is called the *tested language* of T and written $L_{T,IUT}$. We have that $L_{T,IUT}$={L_1, L_2, …, L_n}where L_i =

$L_{ti,IUT}$. Let $Lmax_{T,IUT}$ be the set of tested words in $L_{T,IUT}$ such that none of these words is a prefix of another.

We start by proving a lower bound to the number of input sequences needed to apply all tests in T to a given IUT.

***Proposition* 1**: $Lmax_{T,IUT}$ is exactly the minimum set of input sequences that should be sent to the IUT in order to apply all the tests in T.

Proof: By definition, any word w of the tested language $L_{T,IUT}$ is either in $Lmax_{T,IUT}$ or is a prefix of at least one word in $Lmax_{T,IUT}$.
If w is in $Lmax_{T,IUT}$, then clearly applying all words of $Lmax_{T,IUT}$ will give the result of applying w.
Otherwise, let p be a word of $Lmax_{T,IUT}$ such that w is a prefix of p. Such a word exists by definition of $Lmax_{T,IUT}$. The result of applying w can be inferred from the application of p: since the IUT is deterministic, applying w will produce exactly the result of applying the $|w|$ first inputs of p.
This shows that applying $Lmax_{T,IUT}$ is sufficient. It is also necessary because by definition, any word in $Lmax_{T,IUT}$ is not contained in any other word of the tested language $L_{T,IUT}$, so the result of applying that word cannot be inferred from the application of any tests in T.

Let I be the number of inputs sent to an IUT during the application of T to the IUT and let $|w|$ be the number of inputs in an input sequence w. A lower bound $Min_{T,IUT}$ for I can be easily deduced from the Proposition 1:

***Proposition* 2**: The minimum number of inputs that are needed in order to apply all the tests in T to an IUT is:

$$Min_{T,IUT} = \sum_{w \text{ in } LmaxT,IUT} |w|$$

3. Algorithm

We now present an algorithm that achieves the lower bound on the number of inputs proved in Proposition 2. According to Proposition 1, in order to minimize the number of input sequences sent to the IUT, we need to apply the tests that are going to produce tested words that are not prefixes of any other word in $L_{T,IUT}$, i.e. we need to apply $Lmax_{T,IUT}$. The results for all the other tests can be inferred from these tests.

The problem is that we do not know a priori what tests are going to produce tested words that are not prefixes of any other word in $L_{T,IUT}$ before applying the tests in *T* to the IUT. Instead of actually choosing a particular subset of tests from *T* to apply one after the other, our solution is to apply a group of tests together, and automatically converge toward the application of the subset of *T* that produces the words of $Lmax_{T,IUT}$.

The algorithm works with a set of tests called *currentSet*, which contains all the tests that have the same prefix of the tested word being formed. This constitutes the tests that are currently being considered. Initially, we select all the tests that have the same first input. We send that input to the IUT, observe the response *r* of the IUT and trace this response in all the tests in currentSet by following in each of the corresponding trees the outgoing edge labeled *r*, thus moving to the next node. We then look at the second input of the tests, and again select a subset of tests that have the same second input. We proceed until we have sent the last input of the last test in currentSet. We then select the next set of tests to form the new currentSet.

In the algorithm, we use a vector *V* to store several sets of tests. At any moment, all the tests stored at an index *i* of *V* have the same input sequence consisting of the *i*-1 inputs already sent to the IUT. We call that common input sequence the *prefix* of *V[i]*. We also call *sorting V[i]* ordering the tests stored *V[i]* in ascending order of their next input (i^{th} input). Initially, we store all the tests in *T* in *V*[1]. This is compatible with our definition since all tests have initially the same input sequence of size 0, i.e. no input has been sent yet. Sorting *V*[1] is thus sorting all the tests in T in ascending order of their first input. The algorithm is detailed in Figure 2.

Analysis of the algorithm

The correctness of the proposed algorithm directly follows from Proposition 1, saying that applying the words of $Lmax_{T,IUT}$ is equivalent to applying all the tests in *T*. We are indeed applying the words of $Lmax_{T,IUT}$ because each time we go through the outer while-loop (line 3), we reset the IUT and start a new test. By construction, we carry out this test until the tested word cannot possibly be extended. We are therefore generating maximal tested words, *i.e.* words of $Lmax_{T,IUT}$. Moreover, a test is marked as "finished" in only one case (line 16), when the test actually finishes. Since we carry out the computation until all tests are marked as finished, we cannot possibly miss any word of $Lmax_{T,IUT}$.

Our algorithm is optimal in terms of the number of inputs by virtue of Proposition 2, since we have shown that we stay within the language of $Lmax_{T,IUT}$ and that we completely cover it.

```
 1: index =1;
 2: V[1] = T;
 3: While index $\neq$ 0 do
 4:   If V[index] is not sorted then sort it;
 5:   Reset the IUT and apply the prefix of V[index] to the IUT;
 6:   Let nextSymbol be the next input of the first test in V[index];
 7:   Initialize currentSet with all tests in V[index] having nextSymbol\
                                                    as the next input;
 8:   While currentSet $\neq \emptyset$ do
 9:         Apply nextSymbol to the IUT. Let IUTResponse be the response\
                                                          of the IUT;
10:         index = index + 1;
11:         nextSymbol = null;
12:         For each test t in currentSet do
13:               Trace IUTResponse in t;
14:               If the next input of t == null then
15:                     Mark test t as finished;
16:                     Remove test t from currentSet;
17:               else if the nextSymbol == null then
18:                     nextSymbol = the next input of t;
19:               else if the next input of t $\neq$ nextSymbol then
20:                     V[index] = V[index] $\cup$ t;
21:                     Remove test t from currentSet;
22:               end if
23:         end for
24:   end while
25:   Do
26:         index = index – 1;
27:   until (index == 0) or V[index] $\neq \emptyset$
28: end while
```

Figure 2. Algorithm for testing an IUT using the minimal number of inputs

To analyze the complexity of the algorithm we will consider separately both parts of the algorithm: the sorting part at the beginning of the algorithm and the repetitive part.

Let length (*t*) be the size of the longest path in a test *t*, *w*(*t*) be the maximum number of children of any node in *t*, and |*T*| be the number of tests in *T*.

For the repetitive part, we go through the outer while-loop (line 3) |*T*| times maximum, because we remove at least one test each time. For each iteration of the outer while-loop, we go through the inner while-loop (line 8) at most max(length(*t*)) times where *t* belongs to currentSet, because at each iteration of the inner while-loop, we move to the next input of the set of tests in currentSet. Regarding the complexity of the inner while-loop, line 13 can be done in *O*(*w*(*t*)) for each test *t* in currentSet, while every other step of the loop is done in constant time. Therefore, if length(*T*) is the maximum length(*t*) for *t* in *T* and *w*(*T*) is the maximum of *w*(*t*) for *t* in *T*, the complexity of the repetitive part of the algorithm is bounded by *O*(|*T*|.length(*T*).w(*T*)).

For the sorting, and for a given value of *index*, each test is stored in *V*[*index*] at most once. If we consider that all the tests are sorted at the same time, the sorting can be done in *O*(|*T*|log(|*T*|)). Because index is bounded by length(*T*), the total sorting time is bounded by *O*(length(*T*).|*T*|log(|*T*|)).

The overall complexity of the algorithm is thus bounded by
O(length(*T*).|*T*|.(*w*(*T*) + log(*T*)).

4. Concluding Remarks

The algorithm presented above is optimal in terms of the number of inputs sent to the IUT to apply a set of tests *T*. However, if we make further assumptions regarding the nature of the IUT, the number of inputs can be reduced further.

Resetable IUTs

It is usually assumed that the IUT can only be reset to its initial state. However, in some instances, it is reasonable to assume that the IUT can be reset under some conditions to some or any “recorded” state. For example, the IUT can be a distributed system that can be reset to any stable state (a state where all the sent messages have been received), or the

IUT can be a computer program running under a debugger, where it is sometimes possible to record a state and reset the program to that state at a later stage.

In general, if we assume that any state of the IUT can be recorded and that the IUT can be reset to a recorded state at will and that the IUT implements this reset function correctly, then the algorithm can be improved in the following way:

1) in line 20 of our algorithm, when a test is first stored in V[*index*], the current state of the IUT must be stored as well

2) when the next test is picked up (line 5), instead of resetting the IUT to its initial state and resend the prefix of *V*[*index*], we simply need to directly reset the IUT to the state we now stored along with *V*[*index*].

In that case, the reduction in the number of inputs comes from not sending the prefix of *V*[*index*] to the IUT every time a new currentSet is selected. Instead, we directly reset the IUT to the state where we can start extending the tested word again.

Reversable Units

Another assumption could be that for every input to the IUT, we can send a "reverse input" that will bring back the IUT to its previous state. This is not unlike the "undo" function commonly available in many applications. Under that assumption, the IUT cannot be directly reset to any state, but can be gradually set back to a previous state by sending a reversing sequence. Suppose that the IUT implements these reversing sequences correctly.

Under these assumptions, the algorithm can be modified in the following way: as we find the next branching point (lines 25, 26 and 27), we count how many steps we have to go through. Say we go back from index *k* to index *j*. In other words, we move back from *V*[*k*] to *V*[j]. If k-$j > j$, then instead of resetting the IUT to its initial state (line 5), we send the reverse sequence of inputs to the IUT, from *V*[*k*] to *V*[j].

In other words, once a set of tests in currentSet is completed, we will go back until we find the next set of tests to work with. If that set is relatively close to our starting point (less than half way through to the beginning), it is less costly to “undo” the input sequence in order to bring the IUT to the right state. If the next branching point is further away than half way through, then it is less costly to just reset the IUT to the initial state and bring it to the right state from there.

Acknowledgements

This work was supported in part by Natural sciences and Engineering Research Council of Canada under grant OGP976. The authors thank the anonymous referees for their valuable comments and suggestions of improvements of the presentation of this paper.

References

[1] J. Grabowski, A. Wiles, C. Willcock, and D. Hogrefe. On the design of the new testing language TTCN-3. In Testing of Communicating Systems, 161-176, Ottawa, Canada, Sept. 2000. Kluwer Academic Publishers.

[2] R. M. Hierons and H. Ural. Concerning the ordering of adaptive test sequences. In 23rd IFIP International Conference on Formal Techniques for Networked and Distributed Systems (FORTE 2003), LNCS Volume 2767, 289-302, Berlin, Germany, Oct. 2003. Springer-Verlag.

[3] R. Lai, A survey of communication protocol testing, Journal of Systems and Software, 62(1), 21-46, 2002.

[4] D. Lee and M. Yannakakis. Principles and methods of testing finite-state machines : a survey. Proceedings of the IEEE, 84(8):1089-1123, 1996.

[5] A. Pelc and E. Upfal, Reliable Fault Diagnosis with Few Tests, Combinatorics, Probability and Computing, 7(3), 323 – 333, 1998.

[6] A. Watanabe and K. Sakamura, A specification-based adaptive test case generation strategy for open operating system standards, In the 18th International Conference on Software Engineering, 81–89, Berlin, Germany, May 1996.

[7] M. Yannakakis, Testing finite state machines, In the 23th Annual ACM Symposium on Theory of Computing, 476–485, New Orleans, Louisiana, United States, Jan. 1991.