Postprint of article in *Software: Practice and Experience* 41 (10): 1073–1103 (2011)

Assuring the model evolution of protocol software specifications by regression testing process improvement★*†

Bo Jiang[1], T. H. Tse[1], Wolfgang Grieskamp[2], Nicolas Kicillof[2], Yiming Cao[3], Xiang Li[3], W. K. Chan[4‡]

[1] *The University of Hong Kong, Pokfulam, Hong Kong*
[2] *Microsoft Corporation, Redmond, WA, USA*
[3] *Microsoft Corporation, Beijing, China*
[4] *City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong*

SUMMARY

Model-based testing helps test engineers automate their testing tasks so that they are more cost-effective. When the model is changed because of the evolution of the specification, it is important to maintain the test suites up to date for regression testing. A complete regeneration of the whole test suite from the new model, although inefficient, is still frequently used in the industry, including Microsoft. To handle specification evolution effectively, we propose a test case reusability analysis technique to identify reusable test cases of the original test suite based on graph analysis. We also develop a test suite augmentation technique to generate new test cases to cover the change-related parts of the new model. The experiment on four large protocol document testing projects shows that our technique can successfully identify a high percentage of reusable test cases and generate low-redundancy new test cases. When compared with a complete regeneration of the whole test suite, our technique significantly reduces regression testing time while maintaining the stability of requirement coverage over the evolution of requirements specifications.

KEY WORDS: model-based testing; regression testing; protocol document testing

1 INTRODUCTION

Test engineers automate their testing tasks to make them more cost-effective. Model-based testing is one of the promising approaches to achieve this goal. It verifies the conformance between a specification and the implementation under test (IUT). Test case generation can be automatic, while keeping a clear relationship between requirement coverage and individual test cases. In the model-based testing approach adopted in this work, test engineers first write model programs according to a specification [9]. A *model program*, or simply a *model*, is a formal description of the state contents and update rules for the IUT. Different model programs target different requirements. A set of test cases, also known as a test suite, is automatically generated from each model program for testing.

A specification may evolve during the lifetime of an application when requirements are added, corrected, and removed. The corresponding model program will also be updated to reflect these changes. We refer to the model before the changes as the *original* model program and the one after the changes as the *new* model program.

Because a test suite generated from the original model may not conform to the new model, it is important to revise the test suite effectively to reflect the requirements captured by the new model. A straightforward approach is to regenerate a new test suite from the new model program. This approach, however, is not trivial for complex models. For example, a complete regeneration of the full test suite for a model of a typical protocol testing project in the context of Microsoft's protocol document testing project [9] may take several hours or even a whole day. Moreover, test engineers must then execute all the newly generated test cases and check possibly unaffected features that are irrelevant to the specification change, which may take several days or even one to two weeks to finish at Microsoft.

* This research is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project no. 717308), a Strategic Research Grant of City University of Hong Kong (project no. 7008039), and an internship at Microsoft Corporation.

† A preliminary version of this paper [14] has been presented at the 10th International Conference on Quality Software (QSIC 2010).

‡ Correspondence to: W. K. Chan at Department of Computer Science, City University of Hong Kong, Tat Chee Avenue, Kowloon, Hong Kong. Email: wkchan@cs.cityu.edu.hk.

When determining a test suite, one of the most important goals of the testing team is to achieve a certain level of requirement coverage. The requirements to cover are embedded in the model program and can be tracked during test suite execution. In a typical protocol testing project, it is usually impossible to generate a test suite to achieve full requirement coverage. Test engineers for these specification maintenance projects often aim at achieving high requirement coverage using the generated test cases. Nonetheless, the complete regeneration of the test suite based on a new model may change the requirement coverage drastically. This is due to the nondeterminism inherent in the test case generation algorithm. If the requirement coverage drops a great deal, test engineers have to repeatedly revise the model until the coverage is high enough. On the other hand, if we maximally reuse existing test cases, requirement coverage will be much more stable. Furthermore, as test engineers may have prior knowledge on some existing test cases in terms of requirement coverage, keeping existing test cases in the resultant test suite may assist test engineers in carrying out their testing tasks smoothly. As a result, both researchers and test engineers are seeking solutions that (i) generate test cases targeting at only the features affected by specification change, (ii) maximally reuse existing applicable test cases, and (iii) maintain the stability of the coverage of requirements specifications.

In previous work, Tahat et al. [30] proposed selective regression test case generation techniques to generate regression test cases to test the modified parts of a model. Korel et al. [17] proposed to define a model change as a set of elementary modifications. They further adopted test case reduction strategies to reduce the size of a regression test suite through dependence analyses of deterministic extended finite state machines (EFSM). However, their techniques solved only part of the test suite maintenance problem for model-based regression testing during specification evolution. For instance, when the model evolves, some existing test cases will become obsolete as they only test features that have been removed in the revised specification. It would be a waste of time to execute all of them because these test cases do not help verify the conformance between the new model and the IUT. Moreover, even though the model has evolved, some test cases are still valuable as they test the unchanged parts of the new model. It would save time to identify reusable test cases and avoid the process of regenerating test cases to cover the requirements already covered by the reusable test cases. Furthermore, if the IUT has not changed (but the specification has been updated to reflect correct behaviors of the implementation), as is often the case in protocol document testing at Microsoft, test engineers do not need to rerun reusable test cases to verify the features, thus not only reducing cost but also preserving requirement coverage stability of the test suite during the life cycle of the specification. Third, previous work uses deterministic EFSM to model the behaviors of the IUT. However, many real-life applications are nondeterministic and reactive in nature. To cater for a broader range of applications, we use nondeterministic action machines [11] as the basis of our work.

To address these issues, we propose a REusable Test case analysis technique for mOdel-based Regression Testing (RETORT) to identify reusable test cases and generate a smaller set of new test cases to cover the new model.

Our idea is to apply graph analysis to match every applicable test case generated from the original model with the new model graph. In general, each test case can be considered as an action machine. Intuitively, each test case corresponds to a *graph* of invocations of the IUT and events received from the IUT. Owing to the restrictive purpose of individual test cases, the size of every test case is typically much smaller than that of the original model or the new one. Moreover, the number of test cases going through the subsequent reusability analysis can be controlled, which is crucial to industrial software projects that are subject to cost constraint (whereas if a technique always performs graph comparison between two model programs, part of such a comparison may not be relevant to the test cases that test engineers would like to maximally reuse, resulting in a waste of resources when these parts are large and complex, meaning that the corresponding graph analysis can be lengthy). If RETORT successfully matches every node and the set of outgoing edges of each node specified by a test case on the new model, then the corresponding test case is deemed *reusable*; otherwise, it is deemed as *obsolete*. We note that traditional techniques such as that in [25] only match one edge at a time, which may produce false positive cases. We will discuss them further in Section 5. RETORT also labels all the edges covered by reusable test cases. After that, RETORT builds a subgraph containing all the skipped edges. It will also add outgoing edges of a choice node if any of them have been skipped. Finally, it generates new test cases from the subgraph to achieve edge coverage.

We have implemented our RETORT technique as a new feature of Spec Explorer [9][10], a model-based specification-testing tool built by Microsoft [1]. We apply the technique to the regression testing of four large real-life protocol documents. RETORT is a technique that not only identifies reusable test cases and generates new test cases but also selects test cases from a test suite. Hence, RETORT is multifaceted. From the perspective of identifying obsolete test cases, it is a test case filtering technique. From the perspective of identifying reusable test cases, it is a test case selection technique. From the perspective of producing new test cases, it is a test case generation technique.

This paper extends its conference version [14] in the following aspects:

[1] Spec Explorer is available free of charge at http://tinyurl.com/specexplorer.

1. We report on a rigorous data analysis that verifies the stability of the requirement coverage across a series of evolutions of the protocol documents. To the best of our knowledge, this is the first study on the regression testing of the evolution of real-life protocol models.
2. We extensively elaborate on the results for every evolved version of the protocol.
3. We identify a typical scenario taken from the dataset of the experiment to illustrate how the engineers can, through the use of our technique, improve their model modification styles to accommodate requirement changes.

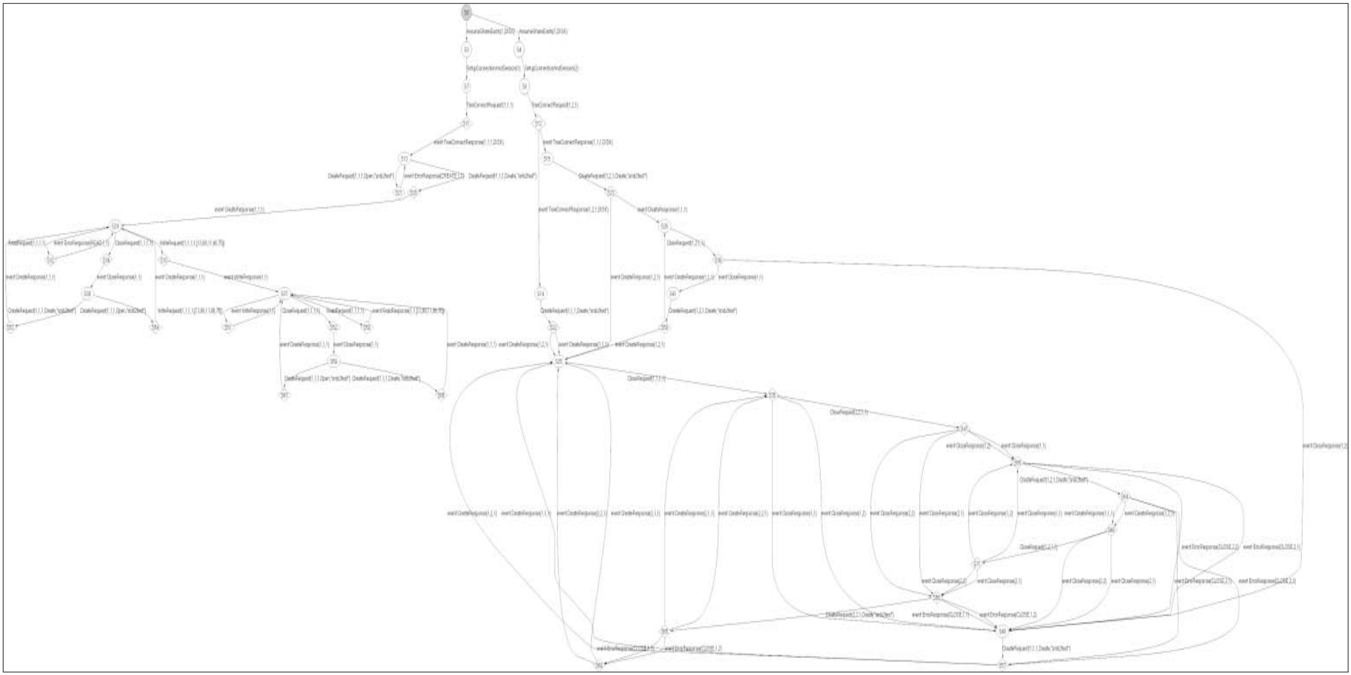


Figure 1. Original model graph

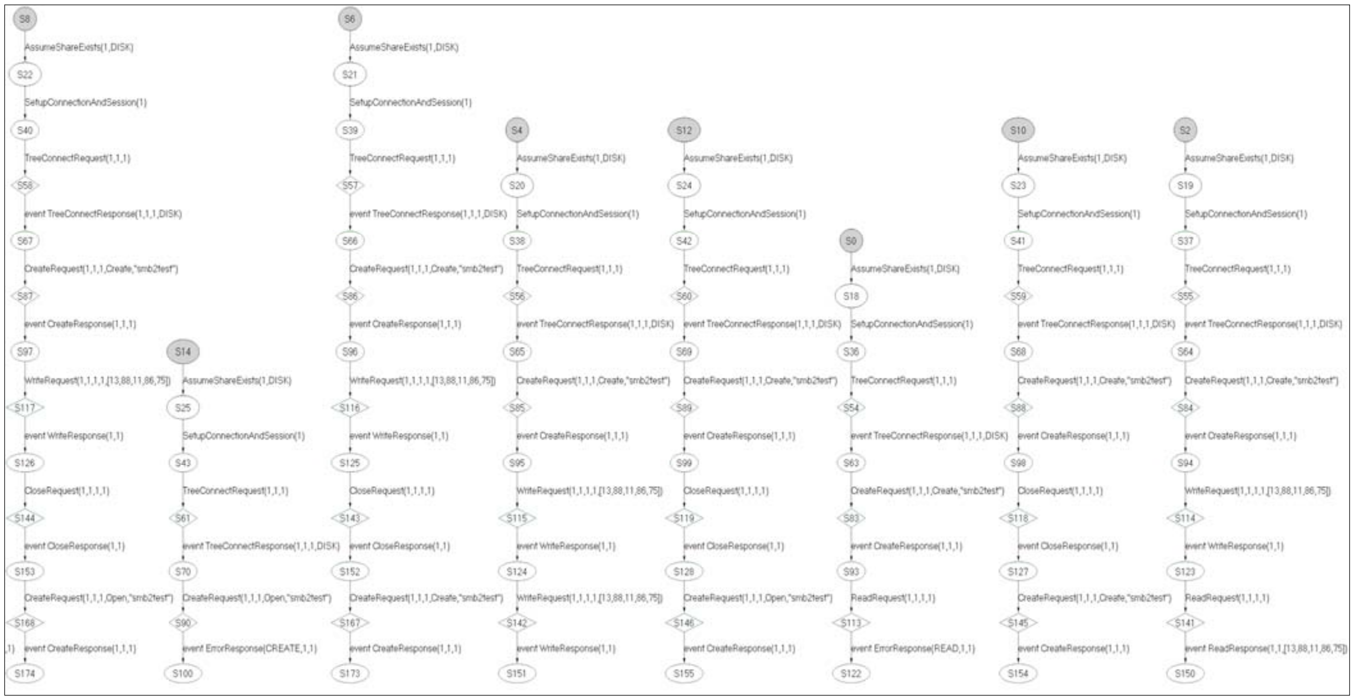

S8
AssumeShareExists(1,DISK)
S22
SetupConnectionAndSession(1)
S40
TreeConnectRequest(1,1,1)
S58
event TreeConnectResponse(1,1,1,DISK)
S67
CreateRequest(1,1,1,Create,"smb2test")
S87
event CreateResponse(1,1,1)
S97
WriteRequest(1,1,1,1,[13,88,11,86,75])
S117
event WriteResponse(1,1)
S126
CloseRequest(1,1,1,1)
S144
event CloseResponse(1,1)
S153
CreateRequest(1,1,1,Open,"smb2test")
S168
event CreateResponse(1,1,1)
S174
S14
AssumeShareExists(1,DISK)
S25
SetupConnectionAndSession(1)
S43
TreeConnectRequest(1,1,1)
S61
event TreeConnectResponse(1,1,1,DISK)
S70
CreateRequest(1,1,1,Open,"smb2test")
S90
event ErrorResponse(CREATE,1,1)
S100
S6
AssumeShareExists(1,DISK)
S21
SetupConnectionAndSession(1)
S39
TreeConnectRequest(1,1,1)
S57
event TreeConnectResponse(1,1,1,DISK)
S66
CreateRequest(1,1,1,Create,"smb2test")
S86
event CreateResponse(1,1,1)
S96
WriteRequest(1,1,1,1,[13,88,11,86,75])
S116
event WriteResponse(1,1)
S125
CloseRequest(1,1,1,1)
S143
event CloseResponse(1,1)
S152
CreateRequest(1,1,1,Create,"smb2test")
S167
event CreateResponse(1,1,1)
S173
S4
AssumeShareExists(1,DISK)
S20
SetupConnectionAndSession(1)
S38
TreeConnectRequest(1,1,1)
S56
event TreeConnectResponse(1,1,1,DISK)
S65
CreateRequest(1,1,1,Create,"smb2test")
S85
event CreateResponse(1,1,1)
S95
WriteRequest(1,1,1,1,[13,88,11,86,75])
S115
event WriteResponse(1,1)
S124
WriteRequest(1,1,1,1,[13,88,11,86,75])
S142
event WriteResponse(1,1)
S151
S12
AssumeShareExists(1,DISK)
S24
SetupConnectionAndSession(1)
S42
TreeConnectRequest(1,1,1)
S60
event TreeConnectResponse(1,1,1,DISK)
S69
CreateRequest(1,1,1,Create,"smb2test")
S89
event CreateResponse(1,1,1)
S99
CloseRequest(1,1,1,1)
S119
event CloseResponse(1,1)
S128
CreateRequest(1,1,1,Open,"smb2test")
S146
event CreateResponse(1,1,1)
S155
S0
AssumeShareExists(1,DISK)
S18
SetupConnectionAndSession(1)
S36
TreeConnectRequest(1,1,1)
S54
event TreeConnectResponse(1,1,1,DISK)
S63
CreateRequest(1,1,1,Create,"smb2test")
S83
event CreateResponse(1,1,1)
S93
ReadRequest(1,1,1,1)
S113
event ErrorResponse(READ,1,1)
S122
S10
AssumeShareExists(1,DISK)
S23
SetupConnectionAndSession(1)
S41
TreeConnectRequest(1,1,1)
S59
event TreeConnectResponse(1,1,1,DISK)
S68
CreateRequest(1,1,1,Create,"smb2test")
S88
event CreateResponse(1,1,1)
S98
CloseRequest(1,1,1,1)
S118
event CloseResponse(1,1)
S127
CreateRequest(1,1,1,Create,"smb2test")
S145
event CreateResponse(1,1,1)
S154
S2
AssumeShareExists(1,DISK)
S19
SetupConnectionAndSession(1)
S37
TreeConnectRequest(1,1,1)
S55
event TreeConnectResponse(1,1,1,DISK)
S64
CreateRequest(1,1,1,Create,"smb2test")
S84
event CreateResponse(1,1,1)
S94
WriteRequest(1,1,1,1,[13,88,11,86,75])
S114
event WriteResponse(1,1)
S123
ReadRequest(1,1,1,1)
S141
event ReadResponse(1,1,[13,88,11,86,75])
S150

Figure 2. Test cases for the synchronous scenario of the model graph

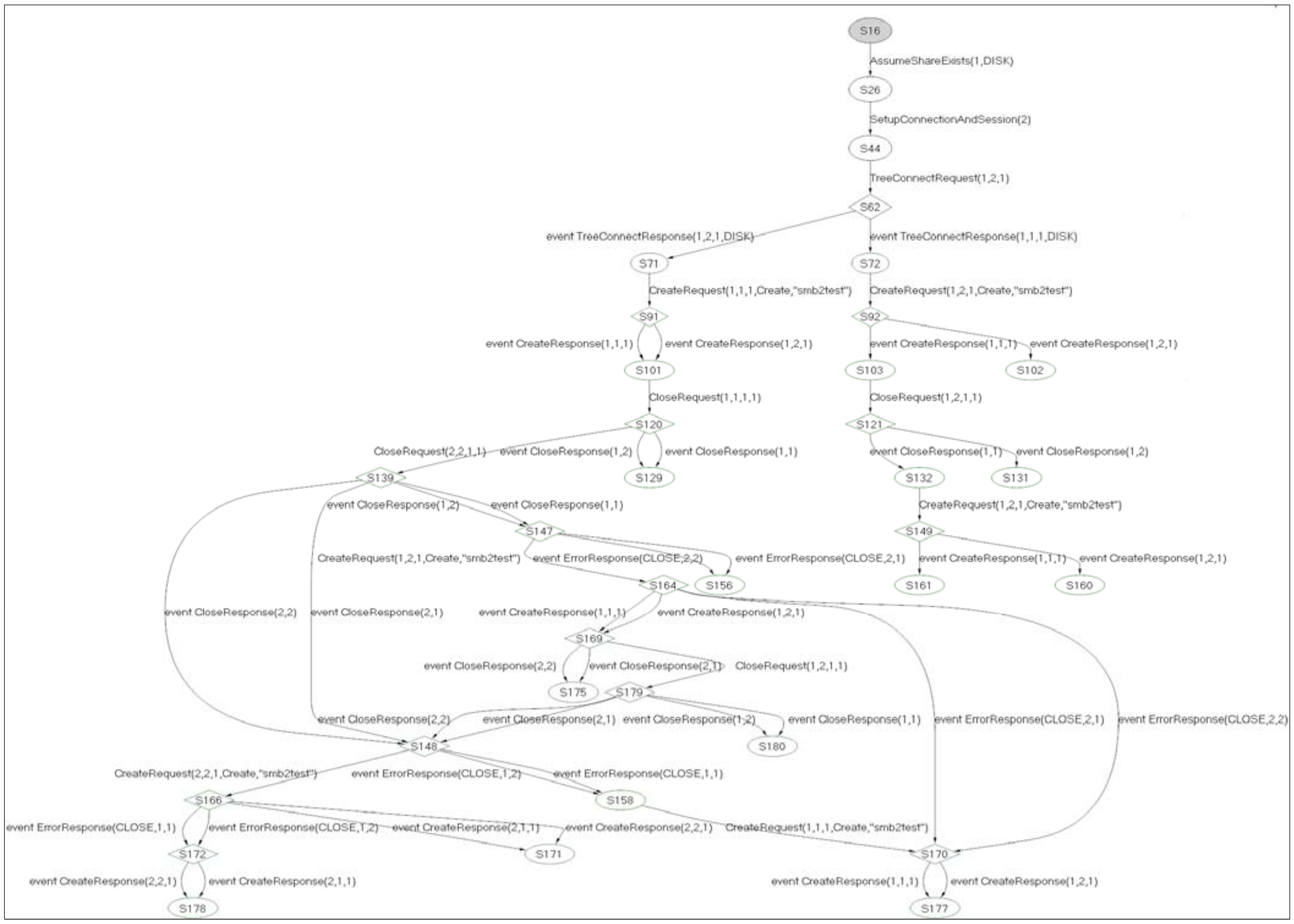


Figure 3. Test case for the asynchronous scenario of the model graph

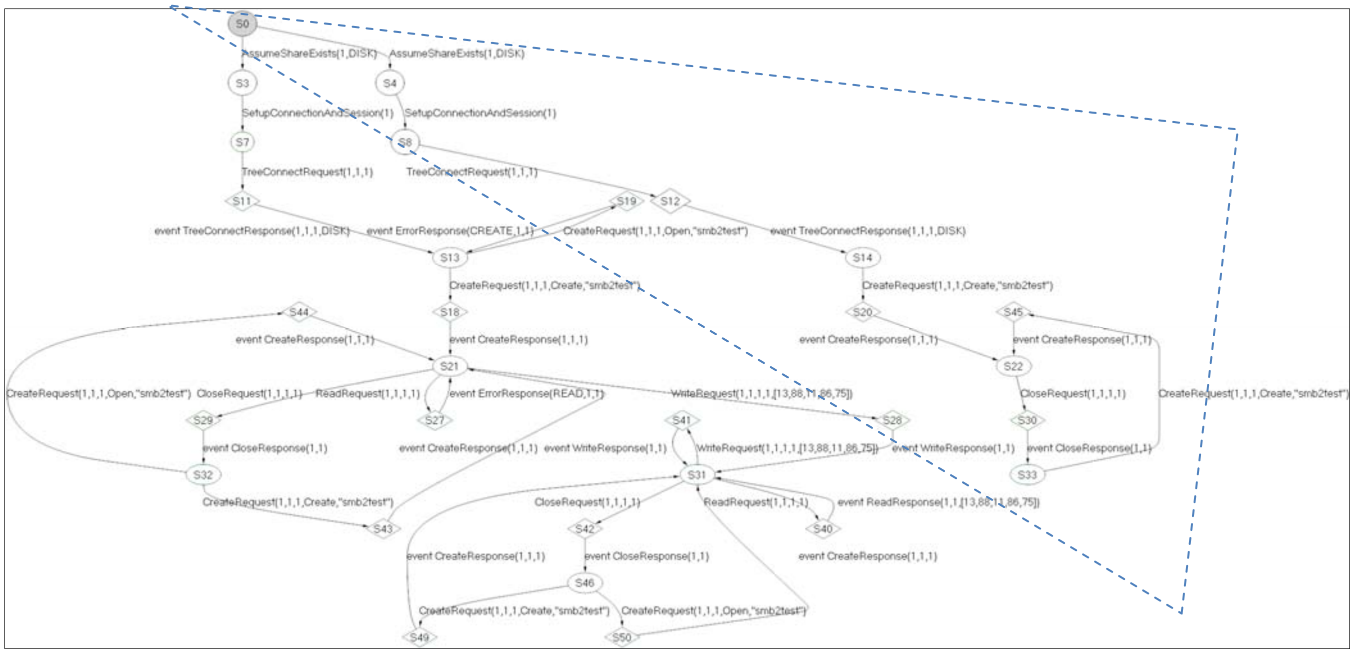
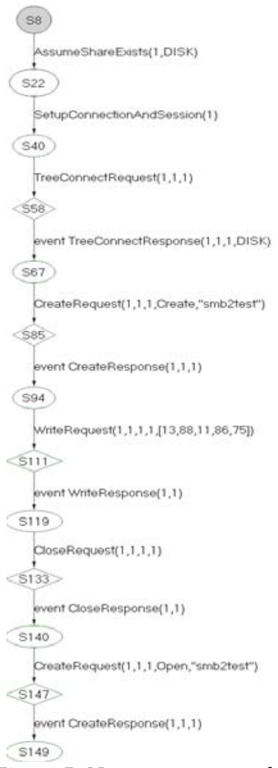


Figure 4. New model graph


S8
AssumeShareExists(1,DISK)
S22
SetupConnectionAndSession(1)
S40
TreeConnectRequest(1,1,1)
S58
event TreeConnectResponse(1,1,1,DISK)
S67
CreateRequest(1,1,1,Create,"smb2test")
S85
event CreateResponse(1,1,1)
S94
WriteRequest(1,1,1,1,[13,88,11,86,75])
S111
event WriteResponse(1,1)
S119
CloseRequest(1,1,1,1)
S133
event CloseResponse(1,1)
S140
CreateRequest(1,1,1,Open,"smb2test")
S147
event CreateResponse(1,1,1)
S149
Figure 5. New test case graph

The contributions of the paper are as follows. First, it proposes a test case reusability analysis and test case generation technique for regression testing of real-world specification evolution. Second, we thoroughly evaluate the proposed technique on the regression testing of four large protocols. The results show that it can successfully identify a large amount of reusable test cases to save regression testing time. Finally, our analysis of the results further advocates good model modification styles for engineers to follow, so that the reusability of the generated test cases can be maximized during regression testing.

We organize the rest of the paper as follows: Section 2 briefly introduces the preliminaries of model-based testing with Spec Explorer. Section 3 presents a motivating example for the proposed model-based regression testing technique that caters for an evolving specification. Section 4 describes in detail the algorithms for the technique. Section 5 presents an empirical study and results analysis. Section 6 describes related work, followed by the conclusion in Section 7.

2 PRELIMINARIES

2.1 Model-based testing with Spec Explorer

In this section, we briefly introduce the process of model-based testing with Spec Explorer for ease of understanding of the subsequent sections.

Test engineers first familiarize themselves with the given specification, and start writing model programs in a mainstream programming language (C#). They define a basic model program M_0 as well as the trace patterns corresponding to test purposes that achieve the desired requirement coverage. A trace pattern is a sequence of events (in terms of the interfaces of IUT) that portrays a valid scenario according to the requirement. The model program is then composed in parallel with each trace pattern in order to reduce the (often huge) state space of the model program. This composition results in a set of submodels of M_0, denoted by M_i (i = 1, 2, ..., n).

Spec Explorer is then used to explore the submodels M_i to generate *model graphs* G_i (i = 1, 2, ..., n) represented by nondeterministic action machines and to generate one test suite from each graph (hence, n is also the number of test suites for the testing project under study). Spec Explorer stores the generated model graphs G_i as intermediate results for test case generation and viewing by test engineers. States are represented in a model graph G_i by two kinds of nodes: *option nodes* and *choice nodes*. If we view the testing of an IUT as a game between a tester and the IUT, an option node represents a state where the tester can make a move by invoking the interfaces provided by the IUT, whereas a choice node represents a state where the tester has to watch and wait for the IUT to take steps. In other words, the outgoing edges of an option node represent actions taken by the tester, while the outgoing edges of a choice node represent actions taken by the IUT. Some option nodes are considered as *accepting*, meaning that the tester's interactions with the IUT can end at that point.

The test case generation step splits each model graph G_i into a set of subgraphs G_{ij} (j = 1, 2, ..., m_i) in test normal form based on the edge coverage criterion, where m_i is the number of test cases in test suite i. A subgraph is in *test normal form* if there is only one outgoing edge for every option node. In other words, steps to be taken by testers are determined. We refer to subgraph G_{ij} as a test case graph because each G_i corresponds to a test suite and each subgraph G_{ij} corresponds to a test case. There are two strategies to generate test case graphs in Spec Explorer: long and short. Both of them try to achieve edge coverage, but the short strategy will stop whenever an accepting node is reached, while the long strategy will try to cover as many edges as possible in every test case. In general, a subgraph generated from the short strategy is much smaller than the model graph, whereas a subgraph generated from the long strategy can be as large as the model graph (provided that it can be traversed in a single path). Furthermore, the size of the subgraph generated by the short strategy is usually much smaller than the model graph, which makes the test cases generated from it more reusable. Finally, Spec Explorer generates an automated test case in C# from each test graph G_{ij}.

2.2 Current practice of regression testing in Spec Explorer

The regression testing support within Spec Explorer is still an emerging feature. When a specification evolves, the original submodel M_i will change to M_i' accordingly. Their corresponding model graphs will also change to G_i'. Traditionally, test engineers using Spec Explorer used to completely abandon all existing test cases and regenerate new test cases for each G_i' by splitting the graph G_i' into new test case graphs G_{ij}'. They then generated a test case from each new test case graph G_{ij}' and executed all of them.

When a change in the specification is small, the difference between M_i and M_i' may also be small (or even non-existing for some i). The corresponding model graph G_i and G_i' will not differ much. Thus, many of the test case graphs G_{ij} may still be valid subgraphs of the new model graph G_i'. In other words, many test cases of the original model may still be reusable for the new model.

2.3 Protocol document testing

In this section, we briefly introduce protocol document testing, which will be our focus in this paper. We may regard protocol documents as specifications of protocol software for interoperability. Protocol document testing [10] is the testing of the conformance between protocol documents and protocol implementations. A protocol implementation sometimes precedes a full interoperability specification by years. As a result, protocol implementations are mature and correct products (verified by their extensive use), whereas protocol specifications may contain errors. Therefore, it is often a fault in the document rather than the IUT that causes a failure in a test. When a fault is exposed, the documents (that is, the specification) are revised, which triggers a modification of the corresponding model and a regression test pass.

Thus, a key difference between protocol document testing and the testing of other applications is that the IUT or the protocol implementation rarely changes during the evolution of the protocol documents. If the IUT has no fault, the reusable test cases will invoke exactly the same call sequences on the implementation and handle the events returned by the implementation in the same manner. In other words, the test results will be the same, and it would be a waste of time to rerun the reusable test cases. For regression testing of protocol documents, therefore, successful identification of reusable test cases is crucial to cost saving.

3 MOTIVATING EXAMPLE

We use a modified version of the SMB2 sample in the Spec Explorer distribution to motivate our regression testing technique RETORT. The sample is a simplified test suite that tests some aspects of the Server Message Block version 2 (SMB2) protocols, which support the sharing of file and print resources between computers.

The testing project uses adapters to expose the interfaces and hide network transmission complexity from the test cases (which act as protocol clients) that interact with an SMB2 server. It includes a model program from which two submodels are selected based on two scenarios: the first scenario is to interact with the SMB2 server synchronously, and the second one is to interact with the server asynchronously (additional requests are sent before previous responses are received). The submodel used for test case generation is the union of the first two submodels. After exploring it, Spec Explorer produces a model graph as shown in Figure 1. We can see that even for a sample testing project, the state space of the model graph is nontrivial. Owing to the limit of a printed page, we cannot show the details of the model graph but only the overall structure. Still, we can see that there are two branches from the start node of this graph: the left branch is for the synchronous scenario, and the right one is for the asynchronous scenario.

In order to generate test cases, Spec Explorer traverses the model graph, splits it into nine test case graphs, and generates C# code from these graphs. The test case graphs corresponding to the synchronous scenario are shown in Figure 2 while the test case graph corresponding to the asynchronous scenario is shown in Figure 3.

Suppose we introduce a model change to the model for the asynchronous scenario, by changing the credit window size from 2 to 1 during the connection and session setup. As a result, the model change will propagate to the model used for constructing test cases. We generate a new model graph from it, as shown in Figure 4. There are still two branches in the new model graph, the one on the left is the unchanged synchronous scenario, and the one on the right, within the triangle with a dotted border, is the changed asynchronous scenario. To make the test suite consistent with the new model, test engineers usually regenerate the whole test suite and then execute it in full. This leads to several problems:

First, even for unchanged parts of the model (such as the synchronous scenario), the newly generated test cases may be different from the reusable test cases because of the impact of the model change. Thus, requirement coverage of the new model will fluctuate owing to the regeneration of the test suite. Second, as test engineers cannot predict the test results and runtime requirement coverage of the new test cases (because of nondeterminism), they need to execute them all, which is time consuming. In the SMB2 example, it takes around 42 min to build the new model graph, generate nine new test cases, and execute all of them.

On the other hand, when we conduct reusability analysis with our RETORT technique, we find that the eight test cases in Figure 2 for the original model are still reusable for the new model. The only obsolete test case is the one in Figure 3. Having identified the obsolete and reusable test cases, we only need to generate new test cases to verify the changed parts of the model, that is, the asynchronous scenario within the triangle in Figure 4. To cater for the parts of the model skipped, only one new test case needs to be generated by RETORT, as shown in Figure 5. Thus, by identifying reusable test cases, we can avoid running them again to save execution time. The total time for RETORT to conduct reusability analysis, new test case generation, and execution of new test case takes only 5 min, which is a great saving compared with the 42 min for the regeneration technique. Furthermore, because the requirement coverage of the reusable test cases is preserved, the requirement coverage of the new test suite can be more stable.

In order to conduct reusability analysis on the test cases of the original model to determine whether they are obsolete or reusable, we start from the initial state of each test case and try to match the labels of the outgoing edges with the labels of the outgoing edges in the new model. These labels are events that trigger the IUT or actions that arise from the IUT. If we can reach the final state of a test case, then it is reusable; otherwise, it is obsolete. For eight of the nine test cases in our example, we successfully match their subgraphs within the new model. However, for the test case in the asynchronous scenario, its test case graph cannot match any part of the new model graph, which makes it obsolete.

During the reusability analysis, we also mark the changed edges as well as the edges solely covered by obsolete test cases. In this way, RETORT produces a subgraph containing the start node, the skipped edges, and the shortest paths from the start node to the skipped edges. It then generates new test cases so that all the edges of the subgraph will be covered. We can also use techniques proposed in previous work to find the impact and side effects on test case generation or reduction due to the changed edges [1][14].

Because the protocol implementation rarely changes in protocol document testing, testers only have to execute the newly generated test cases. Finally, RETORT merges the newly generated test cases and the reusable test cases into a new test suite for the new model for future specification evolution.

4 MODEL-BASED TEST SUITE MAINTENANCE

In this section, we present the key algorithms in detail.

4.1 Test case reusability analysis for model-based regression testing

Before presenting the algorithms, we first show the definitions of the data structures of *Node*, *Edge*, *Graph*, and *NodePair* in Figure 6 to ease discussions. To make the presentation clear, we only show the core structures relevant to the algorithms.

Our test case reusability analysis algorithm is shown in Figure 7. The function *IsReproducible* in the figure performs reusability analysis. It takes four parameters: *testCaseGraph* is the graph of a test case for the original model, *modelGraph* is the new model graph, whereas *testCaseNode* and *modelGraphNode* are the current nodes for comparison in the two graphs. Because our algorithm recursively examines the nodes along the path of *testCaseGraph* and *modelGraph*, the *testCaseNode* and *modelGraphNode* will change dynamically.

While loops on option nodes in a test case graph are removed during test case graph generation, choice nodes can contain loops. We must avoid revisiting such nodes repetitively. At the start of the function *IsReproducible*, we mark the *testCaseNode* as visited by setting its Visited field to be true (line 1). This is checked at line 21 to avoid repeatedly visiting a choice node in the test case graph. In this way, the complexity of our algorithm is bounded by the size of test case graphs.

The function *IsReproducible* performs reusability analysis for two conditions:

The first condition (lines 5 to 22) refers to the case when the node examined is a choice node. In this situation, the IUT may take one of several choices, and the outgoing edges of the choice node in test case graphs should match all the outgoing edges of the choice node in the new model graph. Lines 7 and 8 check whether the number of outgoing edges for the current node agrees in the two graphs. If so, lines 9 to 16 perform a pair-wise match of all outgoing edges in the two graphs. If any outgoing edge fails to match, the test case is identified as obsolete (lines 17 to 19). When all the outgoing edges of a choice node are matched successfully, the algorithm continues to match the respective target nodes of all the outgoing edges (lines 20 to 22).

When all the respective target nodes and their descendants in the test case graph match the new model graph, the whole test case is identified as reusable.

```
// Node in graph
Class Node {
// Outgoing edges of the node
List <Edge> outEdges;
// Incoming edges of the node
List <Edge> inEdges;
// Visiting flag, false by default
bool Visited;
}
// Edge in graph
Class Edge {
// Source node of the edge
Node Source;
// Target node of the edge
Node Target;
// Edge Label
String Label;
}
// Graph base
Class Graph {
// The set of choice nodes in the graph
List <Node> ChoiceNodes;
// The set of option nodes in the graph
List <Node> OptionNodes;
// The set of accepting nodes in the graph
List <Node> AcceptingNodes;
}
// The pair of test case node and model graph node to be compared
Class NodePair {
// The test case node
Node tcNode;
// The model program graph node
Node mpNode;
// Constructor
NodePair(Node tc, Node mp){
tcNode = tc
mpNode = mp;
}
}
```

Figure 6. Data structure definitions

The second condition (lines 24 to 42) refers to the case when the current node to match is an option node. The algorithm first checks whether the node in the test case graph is an end node and the corresponding node in the new model graph is an accepting node (lines 25 to 27). If so, the node is matched successfully. Otherwise, the node in the test case graph should have one and only one outgoing edge because a test case graph must be in test normal form.

However, the corresponding node in the new model graph may have more than one outgoing edges. The algorithm tries to match the outgoing edge of the test case node with each of the outgoing edges of the model graph node (lines 29 and 30). Whenever two edges match successfully, they are added to the set of *coveredEdges* (lines 15 and 31). The algorithm then continues to recursively match the target node of the outgoing edge in the model graph with the target node of the outgoing edge in the test case graph (lines 33 and 34). If two target nodes and their descendants match recursively, the option node is tagged as a match. However, if the two target nodes or any of their descendants fail to match, the algorithm will continue to try and match the outgoing edge of the node in the test case graph with other outgoing edges of the node in the new model graph. Any falsely remembered edges during the trial are removed from the *coveredEdges* set (lines 36 to 40). Finally, if the algorithm cannot match any of the outgoing edges of the model graph, the node is marked as a non-match (lines 41 and 42), which means that the test case is obsolete.

Given a test case graph with m edges and the model program graph with n edges, in the worst case, each edge of the test case is compared with every edge of the model program graph, and so the complexity of the test case augmentation algorithm is $O(nm)$. It should be emphasized that although there may be loops within a test case graph, our technique will not revisit the same node in the test case graph. As a result, the complexity of our algorithm is proportional to the size of the test case graph.

```
testCaseNode: The initial node of a test case graph;
modelGraphNode: The initial node of the model graph;
testCaseGraph: The test case graph;
modelProgramGraph: The model graph;
private bool IsReproducible(testCaseNode, modelGraphNode, testCaseGraph, modelGraph) {
// Mark testCaseNode as visited to avoid repetitive visiting
testCaseNode.Visited = true;
List <Edge> testCaseEdges = testCaseNode.outEdges;
List <Edge> coveredEdges = new List <Edge> ();
List <NodePair> childNodes = new List <NodePair> ();
if (testCaseGraph.ChoiceNodes.Contains(testCaseNode)) {    // Choice node
  if(testCaseEdges.Count != modelEdges.Count)
    return false;
  for each tcEdge in testCaseEdges {
    bool found = false;    // Reset each test case edge
    for each mpEdge in modelEdges {
      if (tcEdge.Label == mpEdge.Label ) {
        found = true;
        // Store the child nodes of both graphs
        childNodes.Add(NodePair(tcEdge.Target, mpEdge.Target));
        coveredEdges.Add(mpEdge);
        break;
      }    // if
    }    // for each mpEdge
    if (! found) {    // At least one edge fails to match, obsolete case.
      childNodes.Clear();
      return false;
    }    // if
  }    // for each tcEdge
  for each (node in childNodes)     // Repeatedly match child nodes
    if (! node.tcNode.Visited && ! IsReproducible(node.tcNode, node.mpNode, testCaseGraph, modelProgramGraph))
      return false;
  return true;
}    // if choice node
else if (testCaseGraph.OptionNodes.Contains (testCaseNode)) {    // Option node, matches any edge of the model graph
  if (testCaseEdges.Count == 0)
    if (modelGraph.AcceptingNodes.Contains (modelGraphNode))
      return true;    // Successfully reproduce
  bool match = false;
  for each (mpEdge in modelProgramEdges) {
    if (mpEdge.Label == testCaseEdges.first().Label) {
      // Try to match along this path
      coveredEdges.Add(mpEdge);
      int lastIndex = coveredEdges.IndexOf(mpEdge);
      // Recursively match children.
      if (IsReproducible(testCaseEdges.first().Target, mpEdge.Target, testCaseGraph, modelProgramGraph)) {
        // Successfully reproduce. Reusable.
        match = true;
        return true;
      }
      else {    // Clear remembered edges when trial fails
        int index = coveredEdges.IndexOf(edge);
        int newEdgeCount = coveredEdges.Count – index;
        coveredEdges.RemoveRange(index, newEdgeCount);
        continue;
    }    // else
    }    // if IsMatch
  }    // for each
  if (! match)
    return false;
}    // else if not choice node
}
```

Figure 7. Reusability analysis algorithm for identification of obsolete and reusable test cases

4.2 Test suite augmentation

Using the test case reusability algorithm, we have partitioned the test suite for the original model into reusable and obsolete test cases. We have also logged all the edges covered by reusable test cases. As we want to achieve edge coverage, we need a test case augmentation algorithm, which generates new test cases to cover all the skipped edges. The algorithm is shown in Figure 8. It starts by finding all the skipped edges of the new model program graph based on all the edges covered by reusable test cases (lines 5 to 7). To cover the skipped edges with new test cases effectively, we first build a shortest path from the initial node of the model graph to the source node of each skipped edge (lines 8 to 11). After that, we combine all the shortest paths to form a subgraph (lines 12 to 17). We also add each skipped edge to the subgraph (lines 18 and 19). Finally, we split the subgraph into new test case graphs in test normal form (lines 20 to 22) and generate the test cases from the new test case graphs to achieve edge coverage (line 23). Given a graph containing *n* nodes and *t* target nodes, the time complexity for the test case augmentation algorithm is $O(tn^2)$.

Note that it is possible for a skipped edge to be an outgoing edge from a choice node. Because we also add the skipped edges to the constructed subgraph, our test case generation algorithm will select all the outgoing edges of a choice node. Thus, the skipped edge will be covered by the newly generated test case.

```
        modelProgramGraph: Model graph of the new model;
        coveredEdges: List containing edges covered by the reusable test cases;
1       TestCaseFile BuildSubgraph(modelProgramGraph, coveredEdges) {
        // The source nodes of all the skipped edges
2       targetNodes = new List <Node>;
3       List <Edge> skippedEdges = new List <Edge> ();
        // New subgraph to cover
4       subgraph = new Graph();
5       for each edge in modelProgramGraph
6         if (! coveredEdges.Contains(edge))
7           skippedEdges.add(edge); // Get the skipped edges
        // Put the source nodes of the skipped edges in targetNodes
8       for each edge in skippedEdges
9         targetNodes.Add(edge.Source);
        // Build shortest paths from start node to all target nodes so that all the skipped edges are reachable from start node
10      ShortestPathAlgorithm spa = new shortestPathAlgorithm(modelProgramGraph, modelProgramGraph.StartNode(),
          targetNodes);
11      spa.Run();
12      for each node in targetNodes {
          // Get the shortest path for each target node and add it to the subgraph
13        path = spa.ResultDict[node];
14        for each newNode in path.Nodes() // Add all nodes
15          subgraph.Add(newNode);
16        for each newEdge in path.Edges() // Add all paths
17          subgraph.Add(newEdge);
        } // for each node
        // Add the skipped edges to the subgraph
18      for each edge in skippedEdges
19        subgraph.Add(edge);
        // Traverse the subgraph to generate test case graph
20      TestCaseAlgorithm testCaseAlgm = new TestCaseAlgorithm (subgraph);
21      testCaseAlgm.Run();
22      newTestCaseGraph = testCaseAlgm.TargetGraph;
        // Generate test case file from newTestCaseGraph
23      return TestCodeGenerator.Generate(newTestCaseGraph);
        }
```

Figure 8. Test case augmentation algorithm

In fact, our work is complementary to previous work on model-based regression test case generation and reduction at this step [1][14]. For example, we may adopt the change impact analysis technique proposed in [1][14] to investigate the impact or side effect of model change on other parts of the model. We can then generate new test cases to cover all affected parts of the model not covered by reusable test cases.

To make the test suite maintenance technique applicable to a succession of model changes, our tool merges the test case graphs of newly generated test cases and reusable test cases to form a test case graph for the current model. We can use this new test case graph for regression testing when the specification evolves again.

5 EVALUATION

In this section, we conduct an empirical study to evaluate the effectiveness of RETORT in supporting specification evolution.

In protocol document testing, test engineers want to cover as much as possible the requirements specified in a protocol specification to gain confidence in the correctness of the protocol documents. In this context, the experiment evaluates RETORT against the regeneration technique (REGEN) with respect to requirement coverage and time costs. Before the development of RETORT, the REGEN technique was the de facto technique used in the protocol document testing project at Microsoft.

5.1 Research questions

***RQ*1.** When the requirements have changed, what percentage of reusable test cases can RETORT identify, and does the test suite generated by RETORT have the property of low redundancy in terms of requirement coverage?

***RQ*2.** When compared with REGEN, how well does RETORT save time costs in regression testing when dealing with a model change?

***RQ*3.** When compared with REGEN, how well does RETORT preserve requirement coverage when handling a model change?

The answer to research question *RQ*1 will tell us whether RETORT has the potential to be useful. If RETORT can only identify a small percentage of reusable test cases, then it will not be worthwhile to use it at all as we still have to regenerate and execute the majority of the test cases. Furthermore, if the new test suite generated by RETORT is so redundant that some test cases just repeatedly cover the requirements already covered by reusable test cases, it is also undesirable to use RETORT as a great deal of effort is wasted on testing the same requirements.

If RETORT is useful in identifying reusable test cases and generating new test cases, we further want to know whether RETORT is indeed useful for practical use. Intuitively, a test engineer is willing to adopt the RETORT technique only if it can save the time cost of testing without compromising the effectiveness.

The answer to research question *RQ*2 will tell us whether RETORT will save time cost. Furthermore, as the requirement coverage of a test suite is an important metric for the effectiveness of protocol document testing, the answer to research question *RQ*3 will tell us whether the effectiveness is indeed maintained.

5.2 Subject programs

We have used four real-life Microsoft protocol document testing projects to evaluate the effectiveness and efficiency of our technique. All subject protocol implementations are production versions in Windows operating systems. The detailed specifications of these protocols are available from the MSDN website [21]. The aim of protocol document testing is to verify the conformance between protocol documents (that is, specifications) and protocol implementations.

Table 1. Subject protocol document testing projects

Subject	**No. of regression versions**	**Total no. of test suites**	**Total no. of test cases**	**Total states**	**Total edges**
BRWS	5	46	448–530	156–170	171–180
CMRP	5	30	1655–1750	47957–47998	67941–67900
COMA	5	211	2310–2402	24577–25300	52816–534899
WSRM	5	45	1487–1510	980–997	1055-1604

The first protocol is BRWS, a Common Internet File System (CIFS) Browser protocol. It defines the message exchange among the service clearing house, the printing and file-sharing servers, and the clients requesting a particular service. CMRP is a Failover Cluster Management API (ClusAPI) protocol. It is used for remotely managing a failover cluster. WSRM is a Windows System Resource Manager protocol, which manages processor and memory resources and accounting functions in a computer. COMA is a Component Object Model Plus (COM+) Remote Administration protocol, which allows clients to configure software components and control the running of instances of these components.

The descriptive statistics of the subjects are shown in Table 1. The column *No. of regression versions* contains the number of modified versions used in the experiments. All of these versions are real modifications of the models made by test engineers, including new action additions, action deletions, action changes, parameter domain changes, and so on. We have obtained these modified versions and their previous versions from the version control repository. The column *Total no. of test suites* shows the total number of test suites across all regression versions for each testing project. Each test suite for the same testing project may cover a different aspect of the requirements. The column *Total no. of test cases* shows the total number of test cases across all test suites and all versions for each testing project. The columns *Total states* and *Total edges* show the total number of states and edges of all the model graphs generated from the model programs of a protocol. The test cases are generated using the strategies specified by the original model program engineers. In this way, we can evaluate our RETORT technique as close to the industrial setting as possible.

The model program development process is as follows: First, the model program developers read the protocol documents and extract all the requirements. They then write basic model programs to define the data structures as well as interfaces exposed by the IUT. After that, they define trace patterns, each of which usually corresponds to a set of requirements to cover. Finally, they consolidate the trace patterns into a model program. This model development process attempts to ensure that the requirements are appropriately covered by the constructed model.

5.3 Experimental environment

We have implemented our regression tool as a stand-alone tool for Spec Explorer 2010 in Visual Studio 2008 Team Suite edition. We have conducted our experiment on a PC with a Pentium 3.0 GHz processor and 2 GB of RAM running Windows 7.

5.4 Experiments and discussions

5.4.1 Experimental Procedure. For each protocol, we first conducted regression testing using the regeneration technique, which abandoned the original test suite, generated new test cases to cover the new model, and executed these test cases. We then conducted regression testing using RETORT, which performed reusability analysis on the test suite of the original model with the algorithm in Figure 7 and generated new test cases with the algorithm in Figure 8, and executed them. For each technique, we measured the time taken to generate the test suite for the new model, the time taken to execute the test suite, and hence the total regression testing time.

5.4.2 Results. In this section, we present the results of our experiment to answer the research questions.

*5.4.2.1 Answering RQ*1. We show in Figure 9 the percentage of test cases identified as reusable by RETORT for each protocol. RETORT finds that, on average, 66%, 52%, 48%, and 33% of the test cases can be reused for a typical version of BRWS, CMRP, COMA, and WSRM, respectively. Thus, RETORT is helpful in identifying reusable test cases.

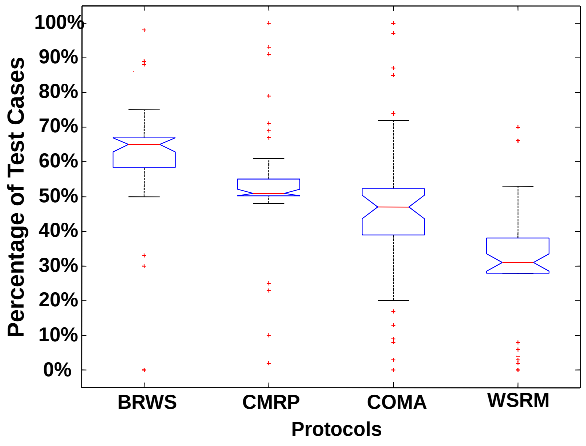


Figure 9. Percentages of test cases identified as reusable by RETORT

The variances of the percentages of reusable test cases range from 18% to 26%, which is not large. Although there are a few outliners, they only account for less than 3% of all the test suites. We have examined every protocol carefully to investigate the cause of the variances. We find that the percentage of reusable test cases varies with the model modification style adopted by the engineers. Different model modification practices have different effects on reusability. We will discuss in Section 5.4.3 how to improve on the model modification style to make test cases more reusable.

The RETORT technique not only identifies reusable test cases but also generates new test cases. We show in Figure 10 the percentage of test cases newly generated by RETORT for each protocol. We find that the mean percentage of test cases thus generated is 39%, 48%, 43%, and 67% for BRWS, CMRP, COMA, and WSRM, respectively. Although there are variances in the mean values, the results indicate that only 39% to 67% of the test cases in the latest test suite need to be executed, which can save a lot of effort. Note that the percentage of test cases is calculated by dividing the number of new test cases by the total number of test cases for the latest version (that is, the number of reusable test cases plus the number of new test cases). Because the percentage of reusable test cases in Figure 9 is computed by dividing the number of reusable test cases by the total number of test cases in the previous version, the sum of corresponding values for Figure 9 and Figure 10 may not necessarily be 100%.

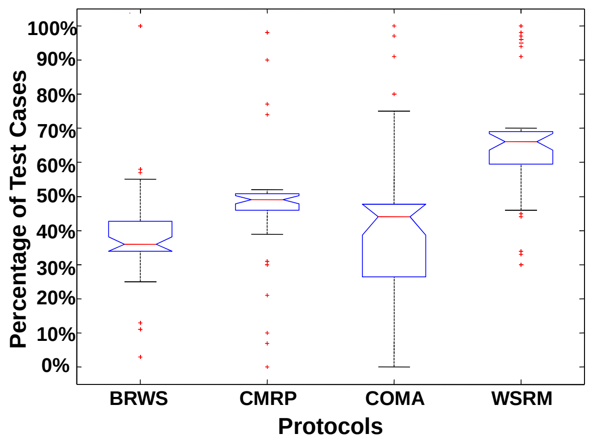


Figure 10. Percentages of test cases newly generated by RETORT

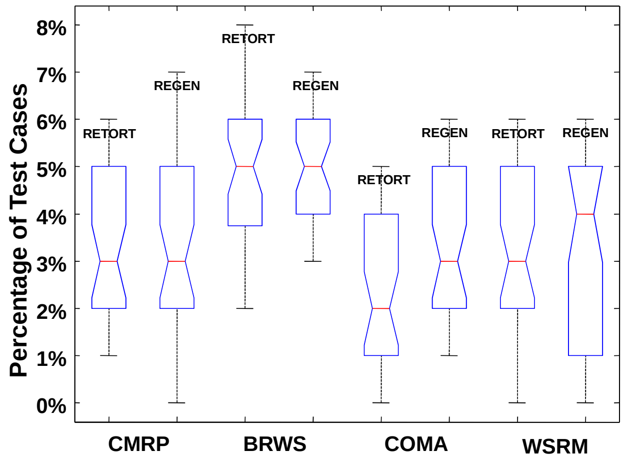


Figure 11. Comparisons of redundant test cases from REGEN and RETORT

As we have pointed out before, a desirable property of the new test suite is that it should have low redundancy. That is, the percentage of multiple test cases covering the same requirements should ideally be small. We calculate the percentage of test cases that are redundant in each test suite and show their distributions in Figure 11. The *x*-axis represents the four different protocols, whereas the *y*-axis shows the percentage of test cases that are redundant in the test suite for each version. There are two boxes for each protocol. The left one represents RETORT, while the right one represents REGEN. We can see that for all protocols, the percentage of RETORT test cases that are redundant is, on average, below 6%, which is low. Furthermore, the redundant test cases from RETORT are always no more than those from REGEN, which means that the new test suites generated by RETORT are no more redundant than the test cases from REGEN.

In conclusion, the answer to research question *RQ*1 is that when the requirements have changed, RETORT can identify a high percentage of test cases that are reusable, and the new test suites generated by RETORT have a desirable property of low redundancy in terms of requirement coverage.

5.4.2.2 Answering RQ2. In this section, we first compare the times for test suite maintenance between RETORT and REGEN. The time cost for test suite maintenance is the total time taken to generate a new test suite according to the new model. For RETORT, it includes the time to conduct reusability analysis for identifying obsolete and reusable test cases (the algorithm in Figure 7) and the time to generate new test cases to cover the subgraph composed from the skipped edges (the algorithm in Figure 8). For REGEN, it includes the time to generate a new model graph from the new model, the time to split the graph into new test case graphs, and the time to generate new test cases from them.

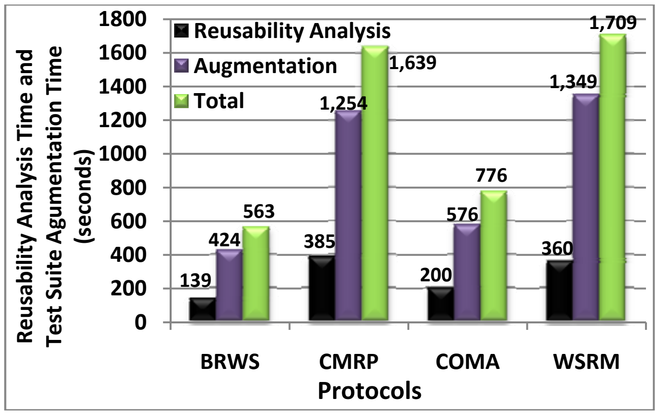


Figure 12. Times for reusability analysis and test suite augmentation by RETORT

Figure 12 shows the time for reusability analysis (using the algorithm in Figure 7), the time for test suite augmentation (using the algorithm in Figure 8), and the total time for test suite maintenance for RETORT. There are three bars for each protocol in Figure 12. The bar on the left shows the time for reusability analysis. The bar in the middle shows the time taken to generate new test cases for test suite augmentation. The bar on the right shows the total time for test suite maintenance, which is the sum of the two bars on the left.

We find that across all protocols, the time used in reusability analysis to identify reusable and obsolete test cases is much smaller than the time used for generating new test cases for test suite augmentation. Because reusability analysis is conducted on all test cases in the test suite while test suite augmentation is only performed on new test cases, the average time for reusability analysis per test case is much smaller than the time taken to generate a new test case. Test suite augmentation may still take most of the test suite maintenance time even when only a few new test cases are involved.

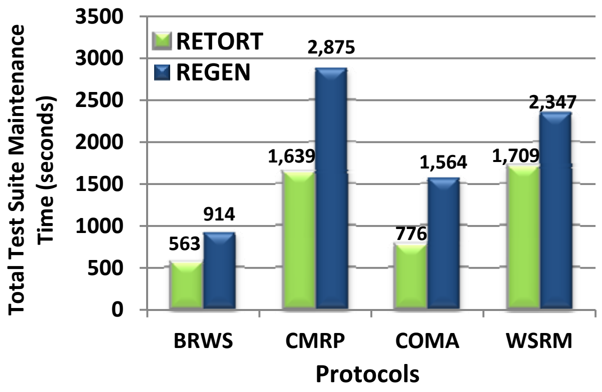


Figure 13. Comparisons of total test suite maintenance times on all test suites between RETORT and REGEN

We then compute the total time for test suite maintenance over all test suites by RETORT and REGEN, respectively, and compare them for each protocol as shown in Figure 13. The *x*-axis in the figure shows the four protocols, while the *y*-axis shows the test suite maintenance time (in seconds) summed up over all suites. There are two bars for each protocol. The bar on the left represents the total test suite maintenance time over all test suites for RETORT, whereas the one on the right represents the corresponding time for REGEN. We find that RETORT requires much less maintenance time than REGEN for every protocol. The total time saving for BRWS, CMRP, COMA, and WSRM is around 351, 1236, 788, and 638 s, respectively.

We have learned from Figure 13 that RETORT can reduce total test suite maintenance time for every protocol. To decide whether the time saving is significant, we conduct an ANalysis Of VAriance (ANOVA) on the test suite maintenance time for each suite to compare RETORT with REGEN, as shown in the notched box-and-whisker plots in Figure 14. The concave parts of the boxes are known as notches. In general, if the notches around the medians in two boxes do not overlap, then there is a significant difference in these two medians at a confidence level of 95%. We see from the figure that for each protocol, the median for RETORT is lower than that of REGEN, and the notches for the RETORT box never overlap with the notches of the REGEN box. Hence, RETORT uses significantly less time for test suite maintenance than REGEN at a 95% confidence level.

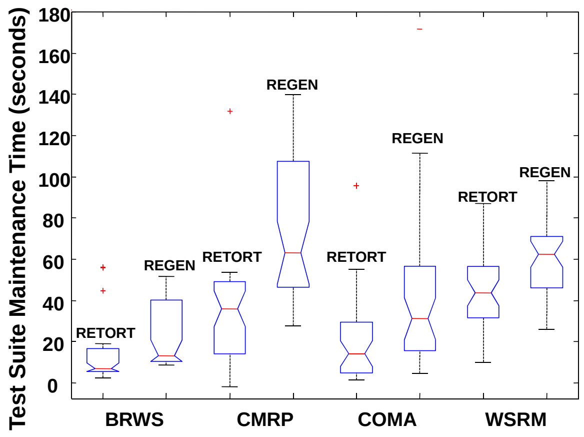


Figure 14. Analyses of variances of test suite maintenance times

We have looked into the test suites in detail to determine why the maintenance time for RETORT is more favorable than that of REGEN. We find that the time taken to analyze the reusability of a test case is only a small fraction of the time taken to generate a new test case. Thus, as long as there are reusable test cases, the time needed to conduct reusability analysis is much less than the time needed to regenerate them. As the final test suite sizes for RETORT and REGEN are almost the same, conducting fast reusability analysis rather than slow regeneration of reusable test cases makes a big difference.

We have found that when compared with the REGEN technique, RETORT reduces the total test suite maintenance time for every protocol. On the other hand, we also want to know how RETORT compares with REGEN in terms of the maintenance time for individual test suites. Are the savings in total test suite maintenance time due to only a few test suites, or are they due to contributions from the majority of test suites? We compare in Figure 15 the savings in test suite maintenance times by RETORT for various protocols. The *x*-axis represents all the test suites for a protocol shown in percentages. The *y*-axis shows the percentage savings in test suite maintenance time of RETORT with respect to REGEN.

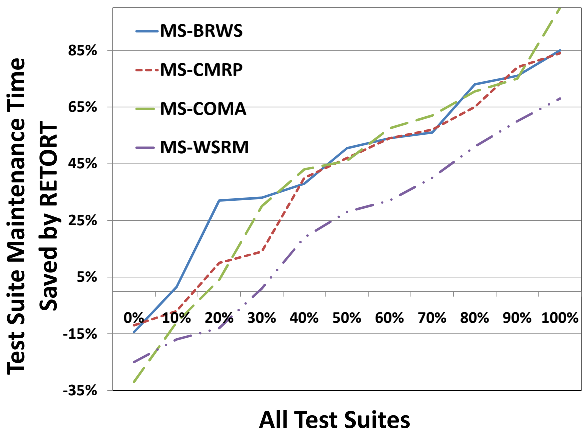


Figure 15. Savings in test suite maintenance times

We observe that when compared with REGEN, the RETORT technique reduces the maintenance time of individual test suites for every protocol. More specifically, we can see from the intersection points of the curves with the *x*-axis that RETORT saves the maintenance times for 90%, 86%, 79%, and 75% of the test suites for BRWS, CMRP, COMA, and WSRM.

After comparing the test case maintenance times between the RETORT and REGEN techniques, we also want to know whether RETORT can save time in test suite execution in the protocol document testing scenario.

For each protocol, we measure the time taken to execute every test suite generated by RETORT and REGEN. Note that we do not have to run reusable test cases in protocol document testing because the IUT is mature and rarely changes. We then compute the total time for test suite execution summed up over all test suites by RETORT and REGEN, respectively, and compare them for each protocol as shown in Figure 16. The *x*-axis in the figure shows the four protocols, while the *y*-axis shows the test suite execution time in minutes. Again, the left-hand bar represents the total test suite execution time for

RETORT, whereas the right-hand one represents the corresponding time for REGEN. We find that RETORT uses much less execution time than REGEN for every protocol. The total time savings for BRWS, CMRP, COMA, and WSRM are 303, 863, 487, and 577 min, respectively.

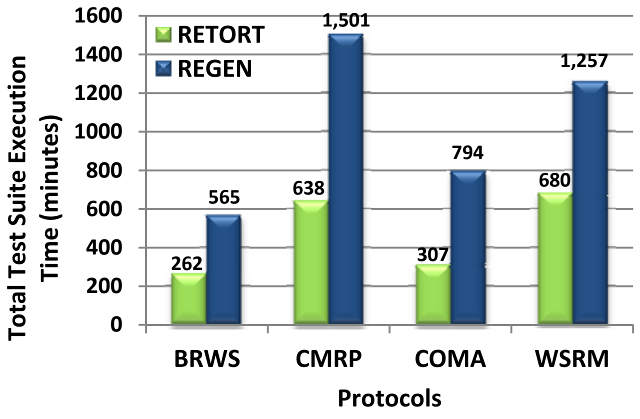


Figure 16. Comparisons of total test suite execution times on all test suites between RETORT and REGEN

We have found from Figure 16 that RETORT can reduce the total test suite execution time for each protocol. Let us further determine whether the time saving is significant. We conduct an ANOVA analysis on the execution time on each test suite for each protocol to compare RETORT with REGEN, as shown in the notched box-and-whisker plots in Figure 17. We can see that for each protocol, the median for RETORT is lower than that of REGEN, and the notches of the RETORT box do not overlap with the notches of the REGEN box. This means that RETORT uses significantly less time for test suite execution than REGEN, at a confidence level of 95%.

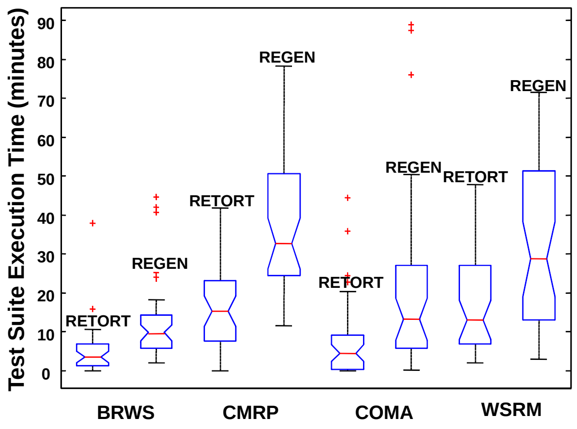


Figure 17. Analyses of variances of test suite execution times

We have looked into the test suites in detail and found that most test cases are reusable. As a result, the test case execution time by RETORT is a small fraction of that by REGEN. For example, one of the test suites for COMA contains 50 test cases. Using RETORT, we successfully identify 35 reusable test cases and take only 8 min to finish the execution of newly generated test cases. However, REGEN takes about 26 min to execute the regenerated test suite.

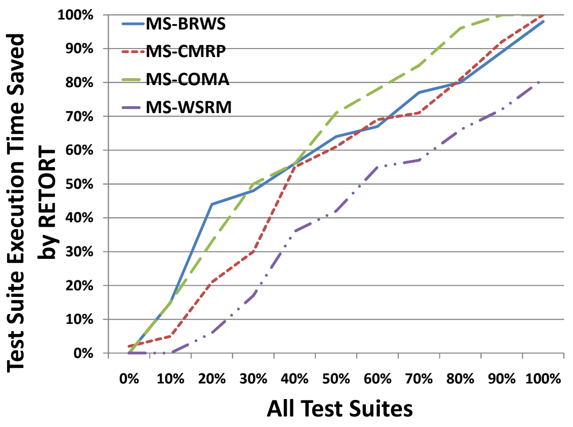


Figure 18. Savings in test suite execution times by RETORT

We have found that when compared with the REGEN technique, RETORT saves the total test suite maintenance time for every protocol. On the other hand, we also want to know how RETORT compares with REGEN in terms of the execution time for individual test suites. Are the savings in total test suite execution time due to only a few test suites, or are they due to contributions from the majority of test suites? We compare in Figure 18 the savings in test suite execution times by RETORT for various protocols. The *x*-axis represents all the test suites for a protocol shown in percentages. The *y*-axis shows the percentage savings in test suite execution time of RETORT with respect to REGEN.

We observe that when compared with REGEN, the RETORT technique saves test suite execution time for every protocol. More specifically, we can see from the intersection points of the curves with the *x*-axis that RETORT saves test suite execution times for 100% of the test suites for BRWS, CMRP, COMA, and WSRM.

After we have compared the test suite execution time between RETORT and REGEN, we continue to compute the total time for regression testing over all test suites by RETORT and REGEN, respectively, and compare them for each protocol as shown in Figure 19. The total regression time is the sum of the test suite maintenance time and the test suite execution time. The *x*-axis in the figure shows the four protocols, while the *y*-axis shows the total regression testing time in minutes. Again, the bar on the left represents the total regression testing time for RETORT, whereas the one on the right represents the corresponding time for REGEN.

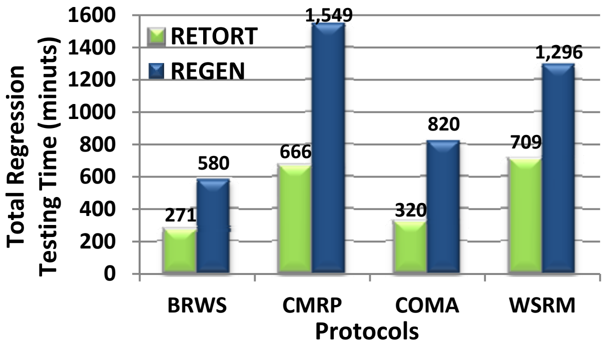


Figure 19. Comparisons of total regression times on all test suites between RETORT and REGEN

Not surprisingly, we find that RETORT uses much less regression time than REGEN for every protocol. The total time savings for BRWS, CMRP, COMA, and WSRM are around 308, 883, 500, and 587 min, respectively.

We have further conducted an ANOVA analysis on the regression testing time on each suite for each protocol to compare RETORT with REGEN, as shown in Figure 20. We observe that for each protocol, the median for RETORT is lower than that of REGEN, and the notches of the RETORT box never overlap with those of the REGEN box. This means that RETORT takes significantly less time for regression testing than REGEN, at a confidence level of 95%.

We have found that when compared with the REGEN technique, RETORT saves the total execution time for every protocol. Are the savings in total execution times due to only a few test suites, or are they due to contributions from the majority of test suites? We compare in Figure 21 the savings in total execution times by RETORT for various protocols. Each percentage saving is computed as the saving in total execution time due to RETORT divided by the total execution time by REGEN.

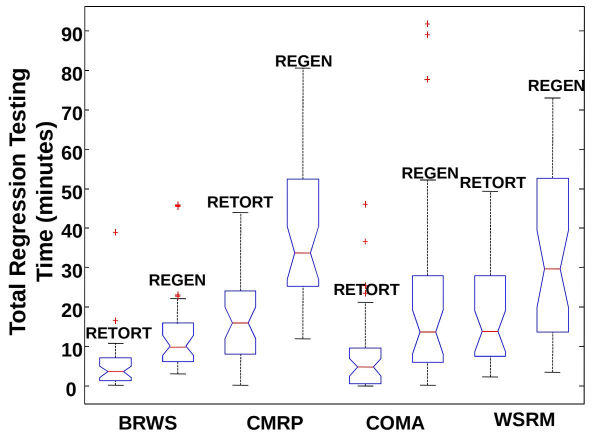


Figure 20. Analyses of variances of regression testing times

We observe that when compared with REGEN, the RETORT technique saves the total execution time for every protocol. More specifically, the intersection points of the curves with the *x*-axis show that RETORT reduces total execution times for almost all of the test suites for BRWS, CMRP, COMA, and WSRM.

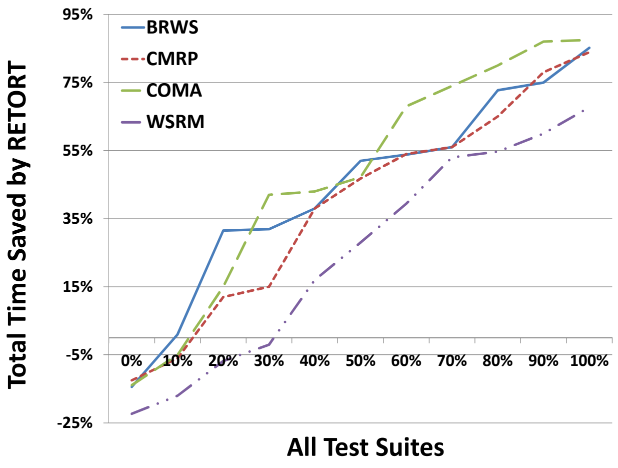


Figure 21. Total time savings by RETORT

Based on the above discussions, our answer to research question *RQ*2 is that when compared with REGEN, the RETORT technique significantly saves test suite maintenance time, test suite execution time, and total regression testing time.

5.4.2.3 Answering RQ3. When a specification evolves, test engineers want the requirement coverage to remain stable despite model changes.

Our experimental results are shown in Figure 22. The *x*-axis lists various test suites for each protocol, whereas the *y*-axis shows the percentage of requirements covered by the reusable test cases in the respective test suite (as against the requirements covered by all test cases in that test suite). Each line in the figure represents a different protocol. From the area between the curve and the *x*-axis, we compute the accumulated percentage of requirement coverage by reusable test cases as against all test cases. For BRWS, CMRP, COMA, and WSRM, the accumulated percentages of requirement coverage are 46%, 49%, 54%, and 41%. Thus, on average, at least 41% of the requirements are covered without even generating and executing any new test cases for the protocols under study. Furthermore, if we measure the requirement coverage of a final test suite (containing both reusable and new test cases) generated by RETORT, it is always approximately equal to that of the corresponding original test suite (with a difference of less than 0.5%).

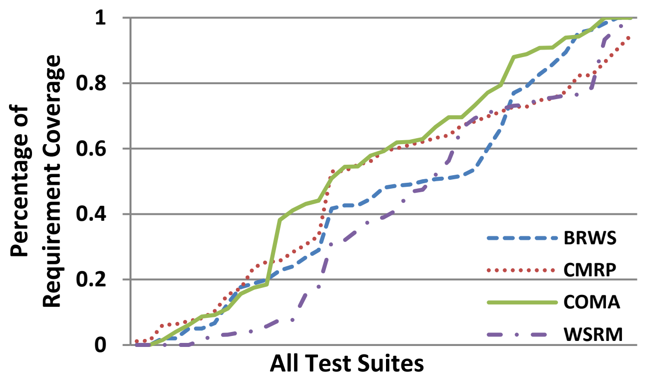


Figure 22. Percentages of requirement coverage of reusable test cases across all test suites

Although reusable test cases identified by RETORT preserve at least 41% of the requirement coverage of the original test suite, one may still have a concern as to whether such coverage is stable across consecutive versions of a protocol. In other words, will RETORT consistently preserve the requirement coverage of the identified reusable test cases across different versions?

We further conducted a case study to explore this problem. We selected six successive versions of each of the four protocols. We collected the percentage of requirement coverage retained by the reusable test cases for each version with respect to its previous version. We show the distributions of the percentages of requirement coverage retained by reusable test cases for various versions of the protocols as box-and-whisker plots in Figure 23 to Figure 26.

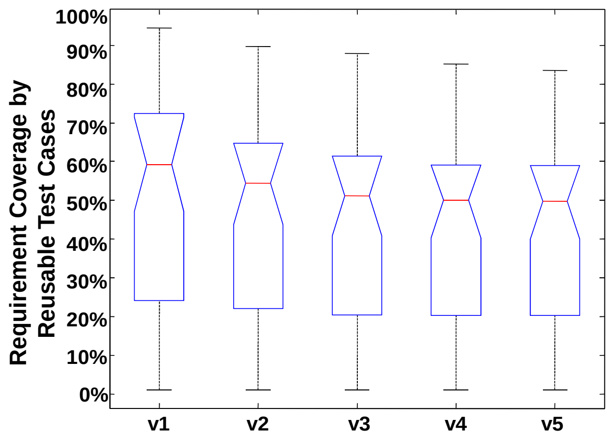


Figure 23. Distributions of percentages of requirements retained by reusable test cases for CMRP

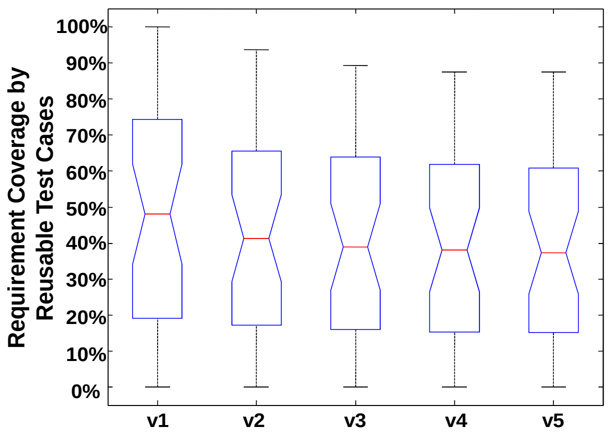


Figure 24. Distributions of percentages of requirements retained by reusable test cases for BRWS

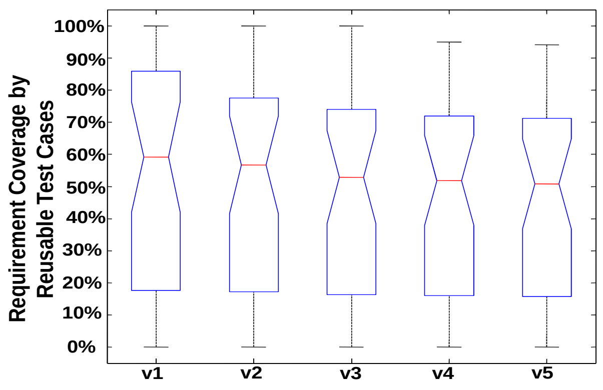


Figure 25. Distributions of percentages of requirements retained by reusable test cases for COMA

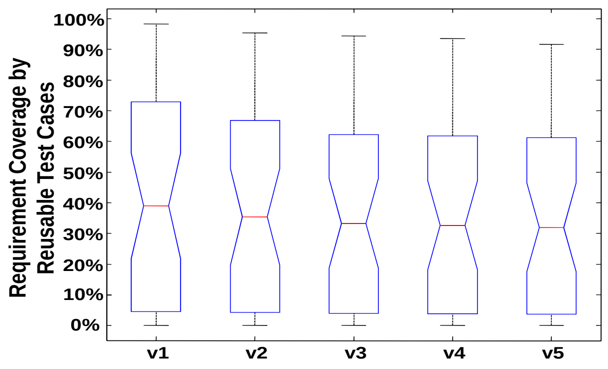


Figure 26. Distributions of percentages of requirements retained by reusable test cases for WSRM

We can see from the plot in Figure 23 that the retained requirement coverage of every version decreases a little with respect to the previous version. However, because the notches of the boxes for all the versions still overlap with one another, the decreases are statistically insignificant. In fact, the decreases become smaller across versions, and the requirement coverage gradually converges. For example, the reusable test cases still cover about 50% of the requirements for version 5 of CMRP. We obtain similar trends in the results for the other three protocols in Figure 24 to Figure 26.

We further investigate individual test cases to see why the requirement coverage retained by reusable test cases is stable across different versions. For example, when we check the CMRP protocol, we find that about 30% of the requirements have never changed across different versions. As a result, the test cases covering them remain consistently reusable and repeatedly identified by RETORT across multiple versions. Furthermore, another 20% of the requirements have never changed between two adjacent versions, while the test cases covering them will also be retained by RETORT.

Thus, our answer to research question *RQ*3 is that RETORT can preserve the requirement coverage of the original test suite to around 40% using reusable test cases only, and close to 100% with the final test suite.

5.4.3 Lessons learned. We have carefully analyzed the results for all test suites and found RETORT to be more effective for some model modifications than others in saving regression testing time. The difference can be related to the modification styles of modelers. Our technique is more effective for model changes in which modelers add new branches to optional nodes to cover a new requirement scenario. In this case, the new branch will generate independent test cases while most existing test cases are still reusable. In contrast, when model changes are made by sequentially appending operations to the current model program, our technique becomes ineffective. This modification style inserts additional nodes to existing paths of the original model graph. Because none of the existing test cases contains these newly added nodes, they invariably fail to reach a final state in the new model graph.

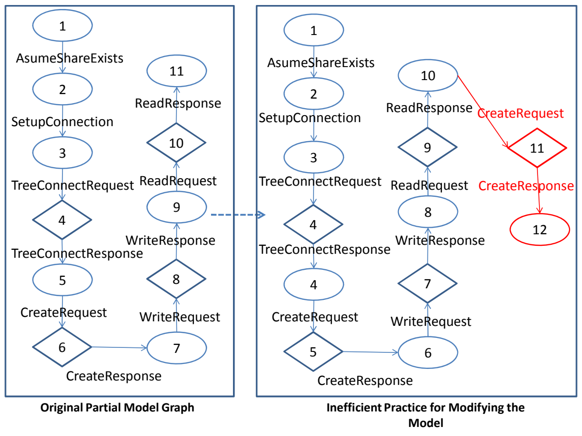


Figure 27. Example of inefficient model modification practice

The example in Figure 27 and Figure 28 illustrates how model modifications may affect the reusability of test cases. The left-hand side of Figure 27 is part of the original model graph for the SMB2 protocol. The graph represents a scenario where a client checks the existence of file sharing on the server (via *AssumeShareExists*), connects to the server (*SetupConnection* and *TreeConnectRequest*), creates a new shared file with a given name (*CreateRequest*), writes data to the created shared file (*WriteRequest*), and reads data back from the shared file (*ReadResponse*). The modification to the protocol specification requires that the server must be able to accept without errors two successive requests to create a new shared file with the same name.

The right-hand side of Figure 27 is an inefficient model modification found in the code repository that keeps the model change history. An engineer has revised the original model to append a new request for creating a new shared file after the read request. As a result, an original test case will invariably become invalid as it will stop after successfully reading back the data and never reach the new end state of the new model graph (that is, the state after receiving the second *CreateResponse*).

In contrast, Figure 28 shows an example of an efficient model modification practice. The left-hand side of Figure 28 shows that the engineer has added the second *CreateRequest* after the option node 7. In this way, we only need to generate and execute a new test case to test the new requirements, as shown in the right-hand side of Figure 28. In this way, the original test case that verifies the requirements of reading and writing of the shared data is still reusable. In fact, the testing of reading and writing of shared data is very time consuming as the protocol needs to read and write a very large shared file from the network. As a result, using the efficient model modification style illustrated in Figure 28, significant regression testing time can be saved by reusing previous test cases. Note that although both modification styles require executing one more test case, the new test case generated from the efficient model modification style takes much less execution time than the one generated from the inefficient model modification style.

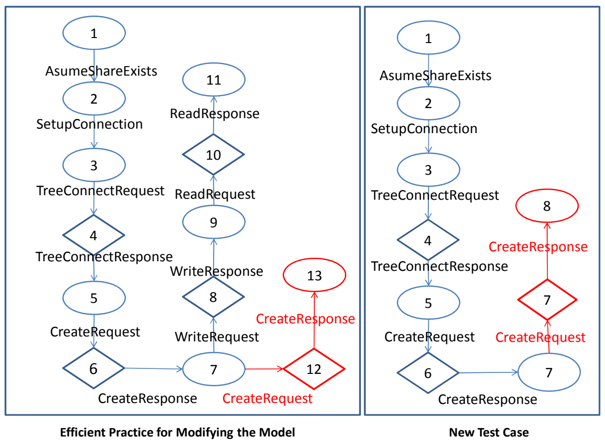


Figure 28. Example of efficient model modification practice

The lesson learned from our findings is that modelers should try to adopt a test-friendly modeling style to enhance the reusability of existing test cases. More specifically, they should try to put different requirement scenarios into different branches of an option node when introducing modifications in the model program. This is similar to what we do in writing unit test cases: we want our test cases to be independent of one another, each covering a different requirement. In this way, we can improve their reusability to save regression time and stabilize requirement coverage.

5.5 Threats to Validity

5.5.1 Internal Validity. We have used several tools in the empirical study, which may have added variance to our results and increased the threats to internal validity. We have applied several procedures to limit the sources of variation.

The model programs in the four protocol document testing projects are developed by the model development team in Microsoft. The engineers are experienced model programmers and have carefully verified that the model programs faithfully reflect the requirement documents.

We tried our best to make our subject programs representative. The subject programs and their test suites are obtained from the version control system of the protocol document testing projects at Microsoft. All the faults exposed by test suites are real protocol document faults logged in the bug fix history of the documents. We have chosen these four subject programs because they are four of the largest testing projects in the repository and because they have relatively complete revision history information. Although our empirical study shows that RETORT is more effective than REGEN on these four subjects, the results may not be generally applicable to all protocol document testing projects. We will evaluate more projects to further evaluate our results in future work.

We have tested our regression testing tools carefully. We developed a systematic test suite to test our testing tools. We tested the tools in several small-scale protocol document testing projects and reviewed their results manually.

5.5.2 External validity. There are several issues that may affect the external validity of our results. The first issue is concerned with the subject programs studied. In this paper, we only evaluate our test technique on the testing of protocol documents whose corresponding IUTs are mature and stable. Although this is common in protocol document testing, it may not be the case for the regression testing of other applications. In fact, our technique is not limited to testing protocol applica-tions only. One may apply the technique to test any applications (such as embedded applications and GUI applications) as long as they have clear requirements specifications and interfaces to interact with. It would be interesting to see the effectiveness of using our regression tools on the testing of applications whose specification and IUT evolve at the same time.

5.5.3 Construct validity. Another threat to validity is that we only measure the time costs and requirement coverage of applying our regression testing technique. This is due to the fact that they are two key concerns of test engineers. However, engineers may also want to know the fault detection capability of the test suite generated by the RETORT technique. A study of this aspect will further strengthen the validity of our empirical study.

6 RELATED WORK

6.1 *Regression test selection techniques*

There are a number of studies on regression test selection. Harrold and Orso [12] gave an overview of the major issues in software regression testing. They analyzed the state of the research and the state of the practice in regression testing, and discussed the major open challenges in regression testing.

One category of these techniques originates from the work of Rothermel and Harrold [25]. They proposed a safe regression test selection approach *DejaVu* based on the impact analysis of modified control flow graphs. Their technique is safe under specific conditions, in which they selected all test cases that have the potential to expose failures. Orso et al. [23] further proposed the use of component metadata for the regression testing of component-based software. They used the same algorithm proposed in [25] to perform both impact analysis and test case selection but took the component metadata information into consideration to improve testing efficiency. Naslavsky et al. [22] used an algorithm similar to that proposed in [25] for test case selection in model-based software. Their work starts from UML sequence diagrams and then transforms them into model-based control flow graphs. Rothermel and Harrold [26] further conducted an empirical study to compare a safe regression test selection technique with the retest-all regression testing strategy. They found that regression test selection is cost-effective but is affected by the test adequacy criteria used to construct the regression test suite. Ball [2] formalized control-flow-based regression test selection techniques using both finite automata theory and intersection graphs. They proposed an edge optimal algorithm to improve the regression test selection algorithm proposed in [25].

Our technique can also be considered as a model-based test case selection technique. Unlike the work reviewed above, ours can be applied to nondeterministic model programs. Let us cite a few basic scenarios to clarify how our work compares with others:

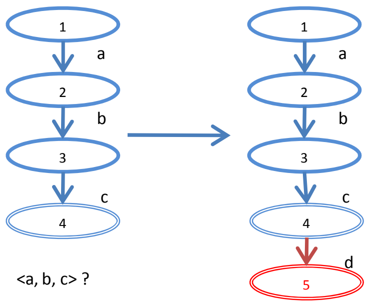


Figure 29. Addition of a new edge to an accepting node

6.1.1 Addition of a new edge to an accepting node. Suppose a model graph is modified by adding a new node, as illustrated in Figure 29. In this scenario, the original model graph is *optionally* appended with a new edge d and a new end state 5. Note that state 4 is still an accepting state, meaning that a test case can either be complete at this state, or optionally take an event d. An existing regression test selection technique such as *DejaVu* [25] identifies the original test case ⟨a, b, c⟩ in Figure 29 as not modification-traversing and does not select it for regression testing. RETORT, on the other hand, identifies this test case as reusable and keeps it for regression testing.

6.1.2 Removal of a non-accepting node. Suppose a model graph is modified by removing a non-accepting node 3 as shown in Figure 30. In this scenario, *DejaVu* identifies the original test case ⟨a, b, c⟩ as modification-traversing and selects it for regression testing. This is because the target node of the edge b changes from a non-accepting node to an accepting node, which makes *all* test cases covering that edge to be selected by *DejaVu*. RETORT, on the other hand, identifies ⟨a, b, c⟩ as obsolete and removes it from regression testing.

6.1.3 Addition of a new edge to a choice node. Suppose a model graph is modified by adding a new edge to a choice node, as illustrated in Figure 31. In this scenario, *DejaVu* identifies the original test case ⟨a, b⟩ as not modification-traversing and removes it from regression testing, but will add a new test case ⟨a, c⟩. RETORT identifies original test case ⟨a, b⟩ as obsolete and removes it from regression testing, but will add a new test case that is prepared to take b or c after a. This is because at state 1, the IUT can return b or c depending on the internal choice made by the implemented program.

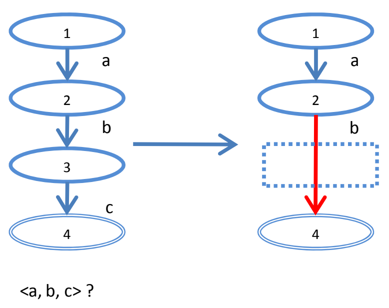


Figure 30. Removal of a non-accepting node

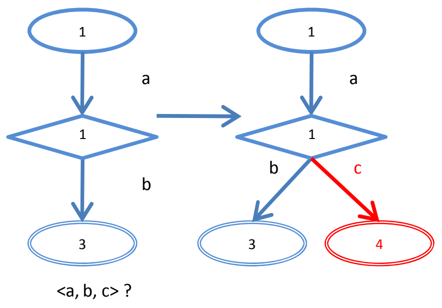


Figure 31. Addition of a new edge to a choice node

From the discussion of the three scenarios, we observe that existing regression test selection techniques may produce false positive and false negative test cases.

Zhang et al. [34] proposed a regression test selection technique by clustering the execution profiles of modification-traversing test cases. It improves existing safe test case selection techniques [25] by further refining the selection process. The results showed that their approach can produce a smaller test suite with a high probability of retaining fault-revealing test cases. However, this technique inherits the limitations of its base technique *DejvVu*.

Chen et al. [5] proposed two regression test selection techniques. The first one targets at conducting regression tests that assures important customer features to be adequately retested. The second technique uses risk analysis to guide test case selection to ensure that potentially high risk problem areas are properly exercised. Their work is based on UML activity diagrams and uses risk information to guide the test case selection process. Briand et al. [3] proposed a technique to classify regression test cases as obsolete, retestable (which must be rerun because of changes), and reusable (which need not be rerun). Their work is based on UML sequence diagrams and class diagrams. As a result, these techniques cannot be applicable to the existing state-based model programs developed in Microsoft.

Farooq et al. [8] presented a methodology for regression test selection using UML state machines and class diagrams. Their approach focuses on finding the impact of a change in class diagrams on state machines and hence on a test suite. Our technique differs from theirs in two aspects: First, we utilized nondeterministic action machines whereas they utilized UML state machines. Second, our technique can generate new test cases to cover those parts of the new model not covered by any original test cases. Their technique mainly partitions existing test suites into resettable, reusable, and obsolete test cases.

Muccini et al. [21] proposed regression test selection techniques to handle the retesting of software systems that both the architecture and the implementation may evolve. Vokolos et al. [31] proposed a regression test selection technique based on textual differences of programs rather than program control flow diagrams or state machines.

6.2 Other regression testing techniques

There are many related studies in model-based testing. Model-based testing allows testers to verify an IUT with respect to a model of the specification. In addition to Spec Explorer, there are other model-based testing tools (such as [13][18]). Korel et al. [17] presented a model-based regression testing approach that uses the dependence analysis of EFSM models to reduce regression test suites. Their approach automatically identifies the difference between the original model and the new model as a set of elementary model modifications. For each elementary modification, they perform regression test reduction accordingly.

El-Fakih et al. [7] also proposed techniques to retest communication software. Our technique differs from theirs in that they worked on FSM models, whereas we worked on action machine model graphs. As a result, their technique cannot handle nondeterminism in choice nodes.

Schieferdecker et al. [28] also proposed regression testing techniques for model-based testing. Their work, however, focuses on the relationship between test cases and the model during the test case generation process based on existing test case selection algorithms. Our work differs from theirs in two aspects: First, they worked on model-based control flow graphs while we used action machine model graphs. Second, similar to the algorithm proposed in [25], their work also results in false positives and false negatives when applied to our scenarios.

Chakrabarti and Srikant [4] proposed the use of explicit state space enumeration to extract a finite state model to compute test sequences to verify subsequent versions of the IUT. Our technique differs from theirs in that we targeted at maximally reusing the original test suite based on the new model graph rather than generating all new test cases from scratch. There have also been many studies on test case prioritization for regression testing. Elbaum et al. [6] presented a family of greedy algorithms, such as the total-statement coverage techniques and the additional-statement coverage techniques. Srivastava and Thiagarajan [29] proposed the computation of program modifications at the basic block level using binary comparison and prioritized test cases to cover the statements thus affected. Walcott et al. [32] studied the test case prioritization techniques under stringent testing time constraints and proposed the use of a genetic algorithm to search for an effective test suite permutation. Li et al. [19] studied various search-based algorithms for test case prioritization with focus on improving code coverage. Their results showed that both genetic algorithms and greedy algorithms were effective in improving code coverage. Zhang et al. [35] further proposed the use of integer linear programming for time-aware test case prioritization. Jiang et al. [15] proposed a family of code coverage-based adaptive random test case prioritization techniques. Their techniques evenly spread the test cases across the input domain to increase the fault detection rate. All these regression testing techniques are hinged on code coverage information. When the coverage information is not available, the techniques cannot be applied in regression testing.

From another perspective, both the code-coverage-based regression testing techniques and our techniques work on graphs. One difference is that they worked on control flow graphs whereas we worked on nondeterministic action machine graphs. It is feasible to further conduct test case prioritization to increase the rate of model coverage or the rate of fault detection on a test suite maintained by our technique (such as through reusable and augmented test cases). It is of practical and research value to study test case prioritization techniques on the test suites thus maintained.

There is also related work on test suite augmentation. Person et al. [24] proposed differential symbolic execution to precisely compute the behavioral characterization of a program change. They also proposed to utilize the program change constraints to identify old (obsolete) and common (reusable) test cases as well as generate new test cases for test suite augmentation. Although their technique works at the program source code level, it will be interesting to extend it to model programs and compare their results with those of our technique. Santelices et al. [27] extended their previous test case augmentation technique based on dependence analysis and partial symbolic execution to identify all change-propagation paths and to handle multiple complex changes. Xu and Rothermel [33] proposed directed test suite augmentation to combine regression test selection and incremental concolic test case generation to achieve good coverage. In contrast, our test suite augmentation technique used graph algorithms to construct a partial graph and then passed it to Spec Explorer to generate test cases to cover the skipped edges.

Memon and Soffa [20] proposed regression testing techniques based on GUI models, which are constructed from static or dynamic GUI structure analysis. They proposed to repair unusable test cases to make them valid for regression testing. An extension of our technique to repair obsolete test cases can be interesting.

A tight integration between the testing and debugging processes is also an interesting topic to explore. In our previous work, we studied how well existing test case prioritization techniques support statistical fault localization [16]. We found coverage-based and random techniques to be more effective than distribution-based techniques in supporting statistical fault localization. Exploring the integration of fault localization with model-based regression testing techniques is a natural next move.

7 CONCLUSION

Model-based testing is effective in systematically testing the conformance between a specification and the IUT. When the requirements specification has evolved, the model must also be updated accordingly. This in turn makes the original test suite obsolete or inadequate. Testers may simply regenerate all the test cases and execute them again, but they may lose the opportunity to save time by utilizing reusable test cases. In this paper, we propose a test case reusability analysis technique known as RETORT for model-based regression testing. It can identify obsolete and reusable test cases effectively, generate new test cases to cover changed parts of the model, and combine the reusable and new test cases to form a new test suite for future regression testing. Our experiment on four large protocol document testing projects shows that RETORT can consistently identify a high percentage of the reusable test cases to significantly reduce regression testing time. Furthermore, the test suites generated by RETORT have a desirable property of low redundancy and can maintain the stability of requirement coverage across different versions. Finally, further analysis reveals a useful modeling practice that enables modelers to modify the models in such a way that the reusability of the generated test cases can be improved.

It will be interesting to extend our technique to handle scenarios where both the specification and the IUT may evolve, and investigate test case prioritization techniques that can increase the fault detection rate for model-based conformance regression testing.

REFERENCES

[1] Alur R, Courcoubetis C, Yannakakis M. Distinguishing tests for nondeterministic and probabilistic machines. *Proceedings of the 27th Annual ACM Symposium on Theory of Computing* (*STOC 1995*), ACM Press: New York, NY, 1995; 363–372.

[2] Ball T. On the limit of control flow analysis for regression test selection. *Proceedings of the 1998 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 1998*), *ACM SIGSOFT Software Engineering Notes* 1998; **23**(2):134–142.

[3] Briand LC, Labiche Y, He S. Automating regression test selection based on UML designs. *Information and Software Technology* 2009; **51**(1):16–30.

[4] Chakrabarti SK, Srikant YN. Specification based regression testing using explicit state space enumeration. *Proceedings of the International Conference on Software Engineering Advances*, IEEE Computer Society Press: Los Alamitos, CA, 2006; Article No. 20.

[5] Chen Y, Probert RL, Sims DP. Specification-based regression test selection with risk analysis. *Proceedings of the 2002 Conference of the Centre for Advanced Studies on Collaborative research* (*CASCON 2002*), IBM Press: Indianapolis, IN, 2002; Article No. 1.

[6] Elbaum SG, Malishevsky AG, Rothermel G. Test case prioritization: a family of empirical studies. *IEEE Transactions on Software Engineering* 2002; 28(2):159–182.

[7] El-Fakih K, Yevtushenko N, von Bochmann G. FSM-based re-testing methods. *Proceedings of the IFIP 14th International Conference on Testing Communicating Systems XIV* (*TestCom 2002*), Kluwer: Deventer, The Netherlands, 2002; 373–390.

[8] Farooq Q, Iqbal MZZ, Malik ZI, Nadeem A. An approach for selective state machine based regression testing. *Proceedings of the 3rd International Workshop on Advances in Model-Based testing* (*A-MOST 2007*), ACM Press: New York, NY, 2007; 44–52.

[9] Grieskamp W. Multi-paradigmatic model-based testing. *Formal Approaches to Software Testing and Runtime Verification* (Lecture Notes in Computer Science 4262), Springer: Berlin, Germany, 2006; 1–19.

[10] Grieskamp W, Kicillof N, MacDonald D, Nandan A, Stobie K, Wurden F. Model-based quality assurance of windows protocol documentation. *Proceedings of the 1st International Conference on Software Testing, Verification, and Validation* (*ICST 2008*), IEEE Computer Society Press: Los Alamitos, CA, 2008; 502–506.

[11] Grieskamp W, Kicillof N, Tillmann N. Action machines: a framework for encoding and composing partial behaviors. *International Journal of Software Engineering and Knowledge Engineering* 2006; 16(5):705–726.

[12] Harrold MJ, Orso A. Retesting software during development and maintenance. *Frontiers of Software Maintenance* (*FoSM 2008*), IEEE Computer Society Press: Los Alamitos, CA, 2008; 99–108.

[13] Jard C, Jeron T. TGV: theory, principles and algorithms: a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *International Journal on Software Tools for Technology Transfer* 2005; **7**(4):297–315.

[14] Jiang B, Tse TH, Grieskamp W, Kicillof N, Cao Y, Li X. Regression testing process improvement for specification evolution of real-world protocol software. *Proceedings of the 10th International Conference on Quality Software* (*QSIC 2010*), IEEE Computer Society Press: Los Alamitos, CA, 2010; 62–71.

[15] Jiang B, Zhang Z, Chan WK, Tse TH. Adaptive random test case prioritization. *Proceedings of the 24th IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2009*), IEEE Computer Society Press: Los Alamitos, CA, 2009; 233–244.

[16] Jiang B, Zhang Z, Tse TH, Chen TY. How well do test case prioritization techniques support statistical fault localization. *Proceedings of the 33rd Annual International Computer Software and Applications Conference* (*COMPSAC 2009*) 1, IEEE Computer Society Press: Los Alamitos, CA, 2009; 99–106.

[17] Korel B, Tahat LH, Vaysburg B. Model based regression test reduction using dependence analysis. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2002*), IEEE Computer Society Press: Los Alamitos, CA, 2002; 214–223.

[18] Kuliamin VV, Petrenko AK, Kossatchev AS, Burdonov IB. The UniTesK approach to designing test suites. *Programming and Computer Software* 2003; 29(6):310–322.

[19] Li Z, Harman M, Hierons RM. Search algorithms for regression test case prioritization. *IEEE Transactions on Software Engineering* 2007; 33(4):225–237.

[20] Memon AM, Soffa ML. Regression testing of GUIs. In *Proceedings of the Joint 9th European Software Engineering Conference and 11th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*ESEC 2003/FSE-11*), *ACM SIGSOFT Software Engineering Notes* 2003; 28(5):118–127.

[21] MSDN. Available at http://msdn.microsoft.com/. Last access: December 3, 2009

[22] Naslavsky L, Ziv H, Richardson DJ. A model-based regression test selection technique. *Proceedings of the IEEE International Conference on Software Maintenance* (*ICSM 2009*), IEEE Computer Society Press: Los Alamitos, CA, 2009; 515–518.

[23] Orso A, Do H, Rothermel G, Harrold MJ, Rosenblum DS. Using component metadata to regression test component-based software. *Software Testing, Verification and Reliability* 2007; 17(2):61–94.

[24] Person S, Dwyer MB, Elbaum S, Pasareanu CS. Differential symbolic execution. *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of Software Engineering* (*SIGSOFT 2008/FSE-16*), ACM Press: New York, NY, 2008; 226–237.

[25] Rothermel G, Harrold MJ. A safe, efficient regression test selection technique. *ACM Transactions on Software Engineering and Methodology* 1997; 6(2):173–210.

[26] Rothermel G, Harrold MJ. Empirical studies of a safe regression test selection technique. *IEEE Transactions on Software Engineering* 1998; 24(6):401–419.

[27] Santelices R, Chittimalli PK, Apiwattanapong T, Orso A, Harrold MJ. Test-suite augmentation for evolving software. *Proceedings of the 23rd IEEE/ACM International Conference on Automated Software Engineering* (*ASE 2008*), IEEE Computer Society Press: Los Alamitos, CA, 2008; 218–227.

[28] Schieferdecker I, Knig H, Wolisz A. (eds.) *Testing of Communicating Systems XIV: Applications to Internet Technologies and Services* (IFIP Advances in Information and Communication Technology 82), Springer: Berlin, Germany, 2002.

[29] Srivastava A, Thiagarajan J. Effectively prioritizing tests in development environment. In *Proceedings of the 2002 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2002), ACM SIGSOFT Software Engineering Notes* 2002; **27**(4):97–106.

[30] Tahat LH, Bader A, Vaysburg B, Korel B. Requirement-based automated black-box test generation. *Proceedings of the 25th Annual International Computer Software and Applications Conference* (*COMPSAC 2001*), IEEE Computer Society Press: Los Alamitos, CA, 2001; 489–495.

[31] Vokolos FI, Frankl PG. Empirical evaluation of the textual differencing regression testing technique. *Proceedings of the 14th IEEE International Conference on Software Maintenance* (*ICSM 1998*), IEEE Computer Society Press: Los Alamitos, CA, 1998; 44–53.

[32] Walcott KR, Soffa ML, Kapfhammer GM, Roos RS. TimeAware test suite prioritization. *Proceedings of the 2006 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2006*), ACM Press: New York, NY, 2006; 1–12.

[33] Xu Z, Rothermel G. Directed test suite augmentation. *Proceedings of the Asia-Pacific Software Engineering Conference* (*APSEC 2009*), IEEE Computer Society Press: Los Alamitos, CA, 2009; 406–413.

[34] Zhang C, Chen Z, Zhao Z, Yan S, Zhang J, Xu B. An improved regression test selection technique by clustering execution profiles. *Proceedings of the 10th International Conference on Quality Software* (*QSIC 2010*), IEEE Computer Society Press: Los Alamitos, CA, 2010; 171–179.

[35] Zhang L, Hou S-S, Guo C, Xie T, Mei H. Time-aware test-case prioritization using integer linear programming. *Proceedings of the 2009 ACM SIGSOFT International Symposium on Software Testing and Analysis* (*ISSTA 2009*), ACM Press: New York, NY, 2009; 213–224.