Proceedings of the 10th International Conference on Quality Software (*QSIC 2010*),
IEEE Computer Society Press, Los Alamitos, CA (2010)

Regression Testing Process Improvement for Specification Evolution of Real-World Protocol Software*†

Bo Jiang, T. H. Tse‡
The University of Hong Kong
Pokfulam, Hong Kong
{bjiang, thtse}@cs.hku.hk

Wolfgang Grieskamp, Nicolas Kicillof, Yiming Cao, Xiang Li
Microsoft Corp.
Redmond, WA, USA and Beijing, China
{wrwg, nicok, yimingc, xiangli}@microsoft.com

***Abstract*—Model-based testing helps test engineers automate their testing tasks so that they can be more cost-effective. When the model is changed due to 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 practice. 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, so that we can generate new test cases to cover only the change-related parts of the new model. Our experiment on four large protocol document testing projects shows that the technique can significantly reduce regression testing time when compared with complete regeneration of the test suites.**

Keywords—model-based testing; regression testing; protocol document testing

I. INTRODUCTION

Test engineers have tried hard to automate their testing tasks to make them more cost-effective. Model-based testing (MBT) is one of the most promising approaches to achieve this goal. Test engineers use model-based testing to ensure the consistency between a specification and the implementation under test (IUT). In the approach to MBT adopted for this work, test engineers first write model programs according to the specification [7]. A *model program*, or simply a *model*, is a 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.

The specification may evolve during the lifetime of application when requirements are added, corrected, and removed. The model program will also be updated to reflect the change. We refer to the model before the change as the original model program and the one after the change as the new model program. Since the test suite generated from the original model may not attest to the new specification, it is important to maintain the test suite to reflect the new model effectively. A straightforward approach is to regenerate a new test suite from the new model program, which is often used in practice. This approach, however, is time-consuming for complex models. For example, a complete regeneration of the full test suite for the model of a typical protocol testing project in the context of Microsoft's protocol documentation testing project [7] may take hours or even a full day. 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 weeks to finish. What is more, test engineers in this project aim at achieving high requirement coverage, which is a measure of the requirements covered by the execution of test cases. But the complete regeneration of the test suite based on the new model may change the requirement coverage drastically for various reasons. On the other hand, if we maximally reuse existing test cases, regression testing can be more effective and requirement coverage can be much more stable. As a result, both researchers and test engineers are seeking solutions that (a) enable them to generate test cases targeting only the features affected by specification change and (b) maximally reuse existing (valid) test cases.

In previous work, Tahat et al. [16] propose selective regression test case generation techniques to generate regression test cases to test the modified parts of the model. Korel et al. [12] propose to define the model change as a set of elementary modifications. They further adopt test case reduction strategies to reduce the regression test suite based on dependence analysis of deterministic Extended Finite State Machines (EFSM). However, their solution solves only part of the test suite maintenance problem for model-based

---

* © 2010 IEEE. This material is presented to ensure timely dissemination of scholarly and technical work. Personal use of this material is permitted. Copyright and all rights therein are retained by authors or by other copyright holders. All persons copying this information are expected to adhere to the terms and constraints invoked by each author's copyright. In most cases, these works may not be reposted without the explicit permission of the copyright holder. Permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works must be obtained from the IEEE.

† This research is supported in part by the General Research Fund of the Research Grants Council of Hong Kong (project nos. 716507 and 717308) and an internship at Microsoft Corp.

‡ All correspondence should be addressed to Prof. T. H. Tse at Department of Computer Science, The University of Hong Kong, Pokfulam, Hong Kong. Tel: (+852) 2859 2193. Email: thtse@cs.hku.hk.

regression testing.

Their work falls short in the following aspects of model-based regression testing: First, if the model evolves, some test cases will become obsolete as they only test nonexistent features of the changed specification. It would be a waste of time to execute all of them. Second, when the model evolves, some test cases are still valuable as they test unchanged parts of the new model. It would save time to identify reusable test cases and avoid regeneration. Furthermore, if the IUT has not changed, as is often the case in protocol document testing, test engineers do not need to rerun the reusable test cases to verify the features, which will not only reduce cost but also preserve stability of the requirement coverage of the test suite. Third, previous work uses deterministic EFSM to model the behaviors of the implementation under test. However, many real-life applications are nondeterministic and reactive in nature. To cater for a broader range of applications, we use nondeterministic Action Machines (AM) [9] rather than EFSM as the basis of our work.

A simple approach is to regenerate test cases for the new model, compare the regenerated test suite with the original, and execute only those test cases not found in the original test suite. This solution has a couple of problems. First, the regeneration of test cases for the new model can be time-consuming. Second, the pair-wise comparison between test cases is also costly because a test case can be complex in format, such as in the form of graphs or in C# code.

To address these issues, we propose a REusable Test case analysis technique for mOdel-based Regression Testing (RETORT) to identify the reusable test cases and generate a smaller set of new test cases to cover the new model. How can we determine whether a test case is obsolete with respect to the new model? The exploration of a model program results in a model graph (representing a state machine). An intuitive approach is to compare the original and the new model graph to find the nodes and edges in the original model that have been modified, and label a test case as reusable if it never reaches these nodes and edges in the original model. On the other hand, the identification of the modified nodes and edges between two graphs is equivalent to the subgraph isomorphism problem, which is NP-complete [6]. Given that the state space of a model graph can be huge, the time-complexity of the intuitive approach can be prohibitive.

Each test case corresponds to a sequence of invocations of the IUT and events received from the IUT. Our idea is to match every invocation/event sequence of the original model in the new model graph by means of graph analysis. If we can successfully match a sequence, then the corresponding test case is reusable; otherwise, it is obsolete. Since a test case only represents a small fraction of the graph space, the time-complexity can be reduced. Meanwhile, we also label all the edges covered by reusable test cases. After that, our technique builds a subgraph containing all uncovered edges. Finally, we generate new test cases from the subgraph to achieve edge coverage.

We implement our RETORT technique as a new feature for Spec Explorer [7][17], a model-based specification-testing tool built by Microsoft[1]. We apply the technique to the regression testing of several real-life protocol documents. The results show that our technique can significantly reduce regression-testing time and maximally maintain the stability of requirement coverage.

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 evolutions. Second, we evaluate the proposed technique on the regression testing of four large protocols. The results show it can identify many reusable and time-consuming test cases successfully to save regression testing time. Finally, our analysis of the results also suggests better 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 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 a results analysis. Section 6 describes related work, followed by the conclusion in Section 7.

II. PRELIMINARIES

A. 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. The model program is then composed in parallel with each trace pattern in order to reduce the (often infinite) state space of the model program. This composition results in a set of sub-models of M_0, denoted by M_i ($i = 1, 2, ..., n$).

Spec Explorer is then used to explore the sub-models M_i to generate *model graphs* G_i ($i = 1, 2, ..., n$) representing nondeterministic state 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 three kinds of nodes: an *option node*, a *choice node,* and an *accepting node*. If we view the testing of an IUT as a game between testers and the IUT, then an option node represents a state where testers can make a move by invoking the interfaces provided by the IUT, whereas a choice node represents a state where testers have to watch and wait for the IUT to take steps. In other words, the outgoing edges of an option node represent actions taken by testers while outgoing edges of a choice node represent actions taken by the IUT. Finally, an

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

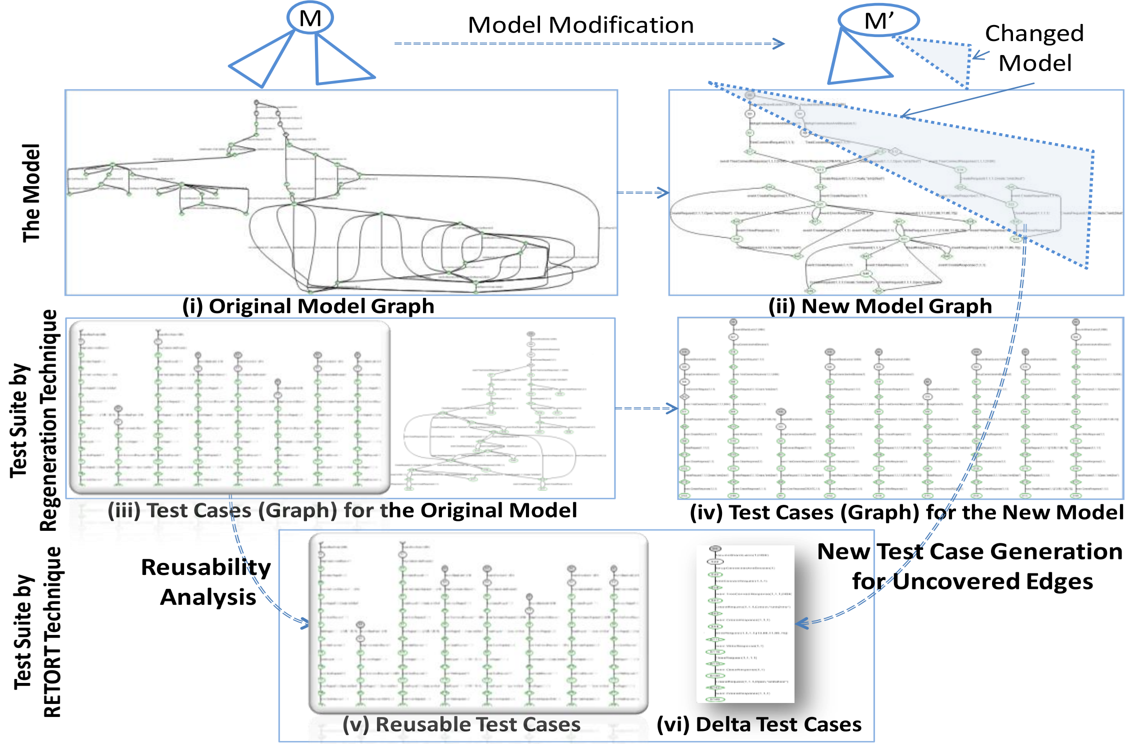

Figure 1. Motivating example: SMB2 protocol testing.

accepting node means that the interactions with the IUT can end.

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 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 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. Finally, Spec Explorer generates a test case in C# from each test graph G_{ij}.

There are two key differences between the state machine model used by Spec Explorer and the EFSM model used by previous work on model-based regression testing. First, Spec Explorer uses action machines (AMs) to represent the update semantics of an action method [9]. The AM framework provides a solid mathematical foundation for handling arbitrarily complex states. In this way, Spec Explorer can use state-based expressions to describe action parameter combinations, and other configurations. Second, the state machine used by Spec Explorer is also nondeterministic in nature as it contains choice nodes that represent nondeterministic behaviors of the IUT. A choice node in a state machine means that the IUT may perform any one of the operations represented by the outgoing edges of the node. For the same test case, the generated test code must be prepared to handle any of these actions taken by the IUT.

B. *Current Practice of Regression Testing in Spec Explorer*

Regression testing support within Spec Explorer is still an emerging feature. When a specification evolves, the original sub-models 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' again by splitting the graph G_i' into new test case graphs G_{ij}'. They then generated the test cases from the new test case graphs G_{ij}' and executed them all.

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, test cases of the original model may still be reusable for the new model.

C. *Protocol Document Testing*

In this section, we briefly introduce protocol document testing, which will be our focus in this paper. We can regard protocol documents as specifications of protocol software for interoperability. Protocol document testing [8] 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 while protocol specifications tend to have faults. 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, modify the corresponding model, and perform regression testing again.

Thus, a key difference between protocol document testing and the testing of other application is that the IUT or the protocol implementation rarely changes during regressions. 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 run the reusable test cases again. For regression testing of protocol documents, therefore, successful identification of reusable test cases is crucial to cost saving.

III. 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) protocol, which supports the sharing of file and print resources between machines.

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 a SMB2 server. It includes a model program from which two sub-models 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. The sub-model used for test case generation is the union of the first two sub-models. After exploring it, Spec Explorer produces a model graph as shown in Figure 1(i). 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, splitting it into nine test case graphs as shown in Figure 1(iii), and generates C# code from these graphs. If we look into the graphs in detail, we find that the eight test cases on the left are for the synchronous scenario while the rightmost one is for the asynchronous scenario.

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 can generate a new model graph from it, as shown in Figure 1(ii). 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, is the changed asynchronous scenario. To make the test suite consistent with the new model, test engineers usually regenerate the whole test suite again as shown in Figure 1(iv) and then execute it in full. This, however, 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 owing to the impact of the model change. Thus, requirement coverage of the new model will fluctuate owing to the regeneration of the test suite. Second, since test engineers cannot predict the test results and runtime requirement coverage of the new test cases (because of non-determinism), they need to execute them all, which is very time-consuming. In the SMB2 example, it takes around 42 minutes 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 eight out of nine test cases in the sample test suite for the original model are still reusable, as shown in Figure 1(v). 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 1(ii). To cater for the uncovered parts of the model, only one new test case needs to be generated by RETORT, as shown in Figure 1(vi). Thus, by identifying the 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 minutes, which is a great saving compared with 42 minutes for the regeneration technique. Furthermore, since the requirement coverage of the reusable test cases is preserved, the requirement coverage of the new test suite will not fluctuate much.

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 it. If we can reach the final state of a test case, then it is reusable; otherwise, it is obsolete. For eight of the test cases in our example, we can successfully match their subgraphs within the new model. However, for the test case in the asynchronous scenario, its test case graph cannot match any parts 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 uncovered edges, and the shortest paths from the start node to the uncovered 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 impacts and side effects on test case generation or reduction due to the changed edges [1][12]. Since 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 as a new test suite for the new model (Figure 1(v) and Figure 1(vi)) for future specification evolution.

IV. MODEL-BASED TEST SUITE MAINTENANCE

We present the key algorithms in detail in this section.

```
    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;
1   private bool IsReproducible(testCaseNode,
      modelGraphNode, testCaseGraph, modelGraph) {
2   testCaseEdges = {outgoing edges of test case node};
3   modelEdges = {outgoing edges of model graph node};
4   childNodes = {child node pair of outgoing edges of test case
      node and model graph node};
5   if (testCaseGraph.ChoiceNodes.Contains(testCaseNode)) {
    // choice node
6     foreach tcEdge in testCaseEdges {
7       bool found = false; // Reset for each test case edge
8       foreach mpEdge in modelEdges {
9         if (match(tcEdge == mpEdge)) {
10          found = true;
            // Store the child nodes of both graph for matching
11          childNodes.Add(tcEdge.Target, mpEdge.Target);
12          coveredEdges.Add(mpEdge);
13          break;
          } // if
        } // foreach mpEdge
14      if (! found) {
        // At least one edge fails to match, obsolete case.
15        childNodes.Clear();
16        return false;
        } // if
      } // foreach tcEdge
17    foreach (node in childNodes)
      // Repeatedly match child nodes
18      if (! IsReproducible(node.tcNode, node.mpNode,
        testCaseGraph, modelProgramGraph))
19        return false;
20    return true;
    } // if choice node
21  else if (testCaseGraph.Nodes.Contains(testCaseNode)) {
    // Not choice node, match any of the edges of model graph
22    if (IsEmpty(testCaseEdges))
23      if (IsAcceptingNode(modelGraphNode))
24        return true; // Successfully reproduce
25    bool match = false;
26    foreach (mpEdge in modelProgramEdges) {
27      if (IsMatch(mpEdge, testCaseEdges.first())) {
        // Try to match along this path
28        coveredEdges.Add(mpEdge);
29        int lastIndex = coveredEdges.IndexOf(mpEdge);
          // Recursively match children.
30        if (IsReproducible(testCaseEdges.first().Target,
          mpEdge.Target, testCaseGraph,
    modelProgramGraph)) {
31          match = true; // Successfully reproduce. Reusable.
32          return true;
          }
33        else { // Clear remembered edges when trial fails
34          int index = coveredEdges.IndexOf(edge);
35          int newEdgeCount = coveredEdges.Count - index;
36          coveredEdges.RemoveRange(index,
    newEdgeCount);
37          continue;
          } // else
        } // if IsMatch
      } // for each
38    if (! match)
39      return false;
    } // else if not choice node
    }
```

Figure 2. Impact analysis algorithm for identification of obsolete and reusable test cases.

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

Figure 3. Test cases augmentation algorithm.

A. *Test Case Reusability Analysis for Mode-Based Regression Testing*

Our test case reusability analysis algorithm is shown in Figure 2. The function *IsReproduceable* performs the reusability analysis. It takes four parameters: *testCaseGraph* is the graph for a test case in the original model, *modelGraph* is the new model graph, and *testCaseNode* and *modelGraphNode* are the current nodes for comparison in the two graphs. Since our algorithm recursively examines the nodes along the path of *testCaseGraph* and *modelGraph*, the *testCaseNode* and *modelGraphNode* will change dynamically.

The function *IsReproducible* performs reusability analysis for two conditions: The first condition (lines 5 to 20) 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 a test case graph should match all the outgoing edges of the choice node in the new model graph. Lines 6 to 13 perform a pairwise match of all the edges in the two graphs. If any outgoing edge fails to match, the test case is identified as obsolete (lines 14 to 16). 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 17 to 19). 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.

The second condition (lines 21 to 39) refers to the case when the current node to match is an option node or accepting node. The algorithm first checks whether the node in the test case graph is an accepting node and the corresponding node in the new model graph is an accepting node (lines 22 to 24). If so, the node is matched successfully. Otherwise, the node in the test case graph is an option node, which 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 26 and 27). Whenever two edges match successfully, they are added to the set of *coveredEdges* (lines 12 and 28). 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 30 and 31). If two target nodes and their descendents match recursively, the option node is tagged as a match. However, if the two target nodes or any of their descendents 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 33 to 37). Finally, if the algorithm cannot match any of the outgoing edges of the model graph, the node is marked as a non-match (lines 38 and 39), which means that the test case is obsolete.

B. Test Case 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. Since we want to achieve edge coverage, we need a test case augmentation algorithm, which generates new test cases to cover all uncovered edges. The algorithm is shown in Figure 3. It starts by finding all the uncovered edges of the new model program graph based on all the edges covered by reusable test cases (lines 4 to 6). Then, to cover the uncovered edges using new test cases effectively, we first build a shortest path from the initial node of the model graph to the source node of each uncovered edge (lines 8 to 12). After that, we combine all the shortest paths to form a subgraph (lines 13 to 19). We also add each uncovered edge to the subgraph (lines 21 and 22). Finally, we split the subgraph into new test case graphs in test normal form (lines 23 to 25), and generate the test cases from the new test case graphs to achieve edge coverage (line 26).

In fact, our work is complementary to previous work on model-based regression test-case generation and reduction at this step [1][12]. For example, we can adopt the change impact analysis technique proposed in [1][12] 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.

V. 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 on the correctness of the protocol documents. In this context, the experiment evaluates RETORT and the regeneration technique (REGEN) with respect to requirement coverage and time costs.

A. Research Questions

***RQ*1.** When compared with REGEN, how well does RETORT save time costs in regression testing when dealing with a model change?

***RQ*2.** When compared with REGEN, how well does RETORT preserve requirement coverage when handling a model change?

The answer to these research questions can tell whether RETORT can be more useful to test engineers than REGEN.

B. Subject Programs

We use four real-life Microsoft protocol document testing projects to evaluate the effectiveness of our technique. All the detailed specifications of these protocols are available from the MSDN website [14]. The aim of protocol document testing is to verify the conformance between protocol documents (that is, specifications) and protocol implementations.

The first protocol is BRWS, the Common Internet File System (CIFS) Browser protocol. It defines the message exchange between the service clearinghouse, printing and file-sharing servers, and clients requesting a particular service. CMRP is the Failover Cluster: Management API (ClusAPI) protocol. It is used for remotely managing a failover cluster. WSRM is the Windows System Resource Manager protocol, which manages processor and memory resources and accounting functions in a computer. COMA is the 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 I. The column *No. of Regression Versions* contains the number of modified versions used in the experiments. All of these versions involve real modifications of the model made by test engineers, including new action additions, action deletions, action changes, parameter domain changes, and so on. We 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.

TABLE I. 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	4	46	448–530	156–170	171–180
CMRP	3	30	1655–1750	47957–47998	67941–67900
COMA	3	211	2310–2402	24577–25300	52816–534899
WSRM	3	45	1487–1510	980–997	1055-1604

C. Experimental Environment

We implemented our regression tool as a standalone tool for Spec Explorer 2010 in Visual Studio 2008 team suite edition. We conduct our experiment on a PC with a Pentium 3.0 GHz processor and 2 GB of RAM running Windows 7.

D. Experiments and Discussions

1) Experiment Procedure. For each protocol, we first perform regression testing with the regeneration technique, which abandons the original test suite, generates new test cases to cover the new model, and executes these test cases. We then conduct regression testing with RETORT, which performs reusability analysis on the test suite of the original model with the algorithm in Figure 2, generates new test cases with the algorithm in Figure 3, and executes them. For each technique, we measure 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.

2) Results.

a) Comparison of Test Suite Maintenance Times between RETORT and REGEN. In this section, we compare the time for test suite maintenance between the RETORT and REGEN techniques. The time cost for test suite maintenance is the total time taken to generate the 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 time to generate new test cases to cover the subgraph composed from uncovered edges, and the time to combine the reusable and new test cases into a new test suite. 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.

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 4. 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 while the one on the right represents the corresponding time for REGEN. We can see that RETORT requires much less maintenance time than REGEN for every protocol. The total time saving for BRWS, CMRP, COMA, and WSRM is around 300 seconds, 1300 seconds, 800 seconds, and 600 seconds, respectively.

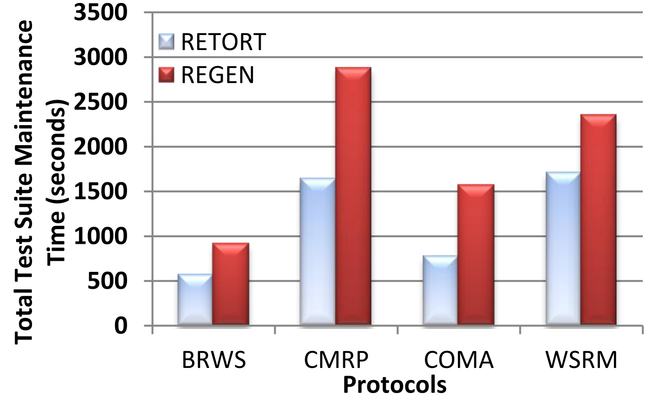

Figure 4. Comparison of total test suite maintenance times on all test suites between RETORT and REGEN.

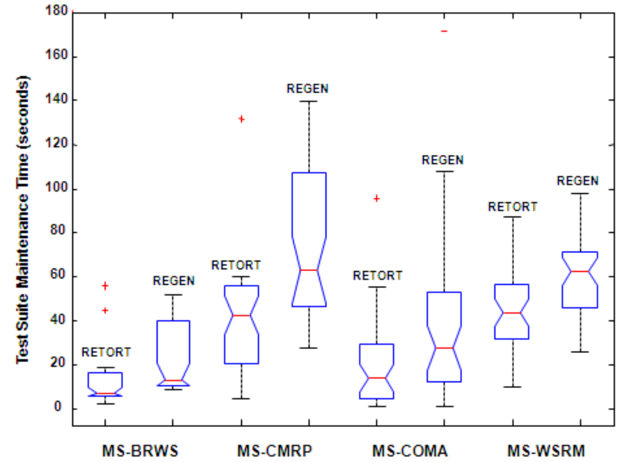

Figure 5. ANOVA analysis of test suite maintenance times.

We know from Figure 4 that RETORT can reduce the 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 5. 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 with a 95% confidence level. 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 confidence level of 95%.

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 have found that the time taken to analyze the reusability of a test case is only about 1/50 of the time taken to generate a new test case. Thus, when there are reusable test cases, the time needed to conduct reusability analysis is much less than the time needed to regenerate them. Since 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.

b) Comparison of Test Suite Execution Times between RETORT and REGEN. Having compared 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.

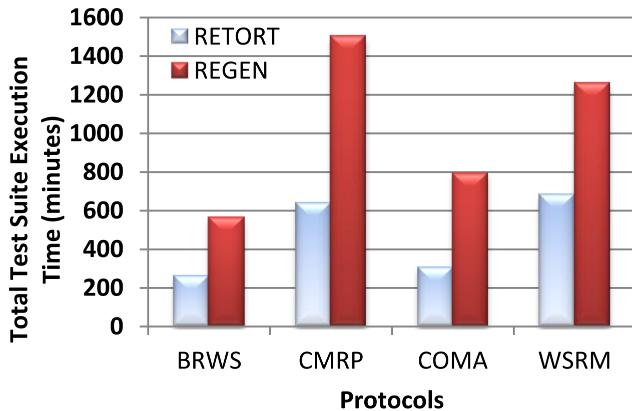

Figure 6. Comparison of total test suite execution times on all test suites between RETORT and REGEN.

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 6. The x-axis in the figure shows the four protocols while the y-axis shows the test suite execution time in seconds. Again, the left-hand bar represents the total test suite execution time for RETORT while the right-hand one represents the corresponding time for REGEN. We can see that RETORT uses much less execution time than REGEN for every protocol. The total time saving for BRWS, CMRP, COMA, and WSRM is around 302 minutes, 862 minutes, 486 minutes, and 576 minutes, respectively.

We have found from Figure 6 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 7. 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%.

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 minutes to finish the execution of newly generated test cases. However, REGEN takes about 26 minutes to execute the regenerated test suite.

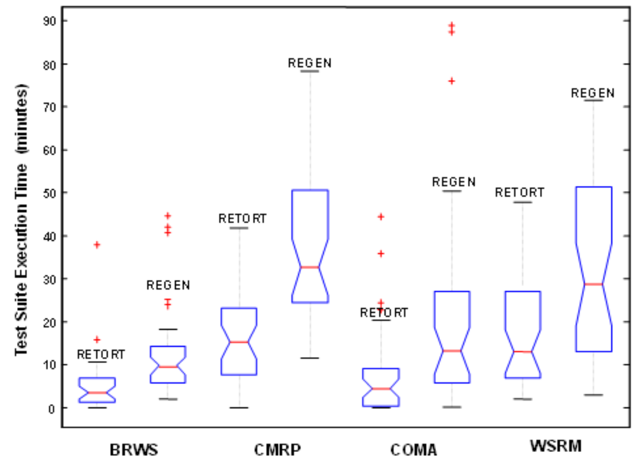

Figure 7. ANOVA analysis of test suite execution times.

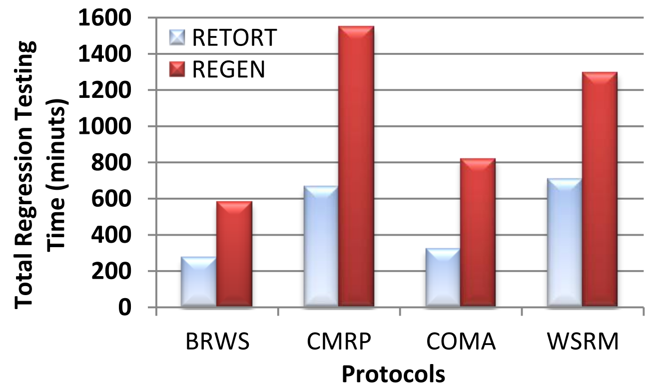

Figure 8. Comparison of total regression times on all test suites between RETORT and REGEN.

c) Comparison of Total Regression Time between RETORT and REGEN. Having compared the test suite maintenance time and test suite execution time separately between RETORT and REGEN, we continue to compare the total regression testing time between them. The total

regression time is the sum of the test suite maintenance time and the test suite execution time.

We then 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 8. 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 while the one on the right represents the corresponding time for REGEN. We can see that RETORT uses much less regression time than REGEN for every protocol. The total time saving for BRWS, CMRP, COMA, and WSRM is around 308 minutes, 883 minutes, 500 minutes, and 587 minutes, respectively.

We further conduct an ANOVA analysis on the regression testing time on each suite for each protocol to compare RETORT with REGEN, as shown in Figure 9. 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 incurs significantly less time for regression testing than REGEN, at a confidence level of 95%.

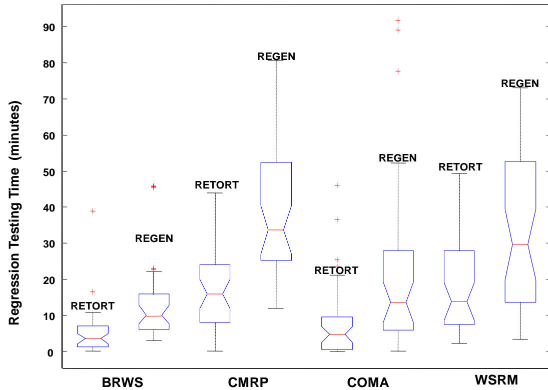

Figure 9. ANOVA analysis of regression testing times.

Based on the above discussions, our answer to research question *RQ*1 is that RETORT can significantly save regression-testing time when compared with REGEN.

a) Requirement Coverage by Reusable Test Cases. When a specification evolves, test engineers want the requirement coverage to remain stable despite model changes.

Our experimental results are shown in Figure 10. The *x*-axis lists various test suites for each protocol while 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 can 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, test engineers can be sure that, on average, around 40% 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.

Thus, our answer to research question *RQ*2 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.

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 is 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 is ineffective. This modification style will insert additional nodes to existing paths of the original model graph. Since none of the existing test cases contains these newly added nodes, they invariably fail to reach a final state in the new model graph.

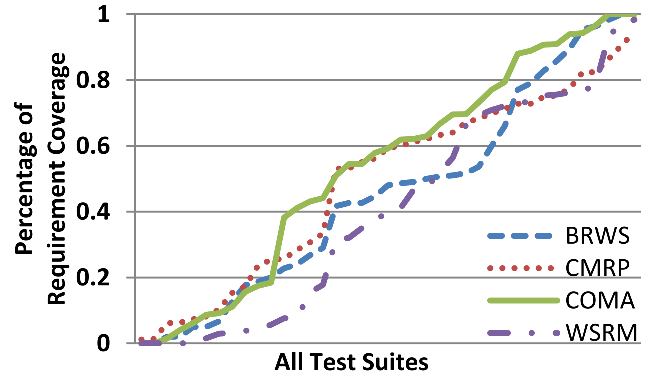

Figure 10. Percentage of requirement coverage of reusable test cases across all test suites.

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.

E. Threats to Validity

In this paper, we only evaluate our test suite maintenance technique on the testing of protocol documents whose corresponding IUTs are mature and stable. A thorough evaluation

of our technique on the regression testing of specification evolutions of other types of applications may further strengthen the validity of our evaluation. Another threat to validity is that we only measure the time costs of applying our regression testing technique. Further measures for the regression fault detection capability of our technique will also strengthen the validity of our empirical study.

VI. Related Work

Model-based testing allows testers to verify an implementation under test with respect to a model of the specification [2]. In addition to Spec Explorer, there are other model-based testing tools [13]. Korel et al. [12] present a model-based regression testing approach that uses EFSM model dependence analysis to reduce regression test suites. Their approach automatically identifies the difference between the original and the new models as a set of elementary model modifications. For each elementary modification, they perform regression test reduction to reduce the regression test suite based on EFSM dependence analysis. El-Fakih et al. [4] and Schieferdecker et al. [15] also propose techniques to re-test communication software. Our work is complementary to theirs in that ours are applicable to the more practical action machines and useful for protocol software.

Chakrabarti and Srikant [3] propose the use of explicit state space enumeration to extract a finite state model to compute good test sequences to verify subsequent versions of the IUT. Our technique differs from theirs in that we target at maximally reusing the original test suite based on the new model graph rather than generate all new test cases from scratch. Farooq et al. [5] present a methodology for regression test case selection using UML state machines and class diagrams. Their approach focuses on finding the impact of changes in class diagrams on state machines and hence on the test suite. Our technique differs from theirs in two aspects: First, we focus on nondeterministic action machines while they focus on 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. Harrold and Orso [10] give an overview of the major issues in software regression testing. They analyze the state of the research and the state of the practice in regression testing, and discuss the major open challenges in regression testing.

VII. Conclusion

Model-based testing is effective in systematically testing the conformance between a specification and the implementation under test. When the 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 significantly reduce the regression testing time and maintain the stability of requirement coverage when compared with a complete regeneration of the whole test suite. 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 implementation under test may evolve, and investigate test case prioritization techniques that can increase the fault detection rate for model-based conformance regression testing.

References

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

[2] M. Barnett, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Towards a tool environment for model-based testing with AsmL. In *Proceedings of the 3rd International Workshop on Formal Approaches to Testing of Software* (*FATES 2003*), volume 2931 of Lecture Notes in Computer Science, pages 252–266. Springer, Berlin, Germany, 2004.

[3] S. K. Chakrabarti and Y. N. Srikant. Specification based regression testing using explicit state space enumeration. In *Proceedings of the International Conference on Software Engineering Advances*. IEEE Computer Society Press, Los Alamitos, CA, 2006.

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

[5] Q. Farooq, M. Z. Z. Iqbal, Z. I. Malik, and A. Nadeem. An approach for selective state machine based regression testing. In *Proceedings of the 3rd International Workshop on Advances in Model-Based testing* (*A-MOST 2007*). ACM Press, New York, NY, 2007.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, New York, NY, 1990.

[7] W. Grieskamp. Multi-paradigmatic model-based testing. In *Formal Approaches to Software Testing and Runtime Verification*, volume 4262 of Lecture Notes in Computer Science, pages 1–19. Springer, Berlin, Germany, 2006.

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

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

[10] M. J. Harrold and A. Orso. Retesting software during development and maintenance. In *Frontiers of Software Maintenance* (*FoSM 2008*), pages 99–108. IEEE Computer Society Press, Los Alamitos, CA, 2008.

[11] C. Jard and T. Jeron. 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*, 7 (4): 297–315, 2005.

[12] B. Korel, L. H. Tahat, and B. Vaysburg. Model based regression test reduction using dependence analysis. In *Proceedings of the IEEE*

International Conference on Software Maintenance (*ICSM 2002*), pages 214–223. IEEE Computer Society Press, Los Alamitos, CA, 2002.

[13] V. V. Kuliamin, A. K. Petrenko, A. S. Kossatchev, and I. B. Burdonov. The UniTesK approach to designing test suites. *Programming and Computer Software*, 29 (6): 310–322, 2003.

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

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

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

[17] M. Veanes, C. Campbell, W. Grieskamp, W. Schulte, N. Tillmann, and L. Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In *Formal Methods and Testing*, volume 4949 of Lecture Notes in Computer Science, pages 39–76. Springer, Berlin, Germany, 2008.