Towards Automating Regression Test Selection for Web Services

Michael Ruth
University of New Orleans
2000 Lakeshore Dr
New Orleans, LA 70148
mruth@cs.uno.edu

Shengru Tu
University of New Orleans
2000 Lakeshore Dr
New Orleans, LA 70148
shengru@cs.uno.edu

ABSTRACT

This paper reports a safe regression test selection (RTS) approach that is designed for verifying Web services in an end-to-end manner. The Safe RTS technique has been integrated into a systematic method that monitors distributed code modifications and automates the RTS and RT processes.

Categories and Subject Descriptors

D.2.5 [Testing and Debugging]: Testing Tools.

General Terms: Algorithms, Design, Experimentation, Management, Reliability, Verification.

Keywords: Regression Test Selection, Web Services, Automation, Control-Flow Graphs

1. INTRODUCTION

Web services must undergo rapid adjustments, since the businesses they support are frequently changing. These modifications must be supported by rapid verification. In addition to the correctness of the new functions, we have to assure that each modification does not impose any adverse effect on the unmodified functions. A common practice is to rely on regression testing (RT). Without a test selection process, regression testing would require retest all the existing test cases. This approach over time becomes less and less affordable for complex systems, when more and more test cases are added to the test suite. One of the key ideas of RT is to reduce the number of tests that need to be retested, or regression test selection (RTS). Compared to other RTS techniques, the safe regression test selection techniques guarantee that no modification-revealing and thus possibly fault-revealing test case will be left unselected and therefore untested [1]. However, there is no available mechanism to apply safe RTS techniques to Web services due to the distributed and autonomous nature of Web services. While each service is thought of its own development island, the services utilize each other to perform complex business functions. This leads to issues rooted in both the functional and verification dependencies between services.

In this paper, we propose a safe RTS technique for verification of Web service systems in an end-to-end manner. Our approach is based on the safe RTS algorithm by Rothermel and Harrold which was developed for monolithic applications using control flow graphs (CFG) [1]. Rather than requiring all the source code of the participating services and applications, we require CFGs from every party. The granularity of the CFGs can vary from very detailed to very abstract. Using hash code, the CFGs will be able to indicate changes but shield the program source code. In this way, we have adopted the safe RTS, originally a white-box technique, into a grey-box technique that can work for inter-enterprise systems. Our approach has been designed to automate the RTS process, and be capable of precisely locating the sources of each fault. We have been developing a framework that monitors distributed code modifications and automates the RTS process as well as the test-running management.

There are quite a few regression testing techniques and tools for generating test cases and performing the regression testing for Web services. Our most closely related works were Tsai's RTS framework that enhances WSDL and uses UDDI [2], and Harrold's RTS for component-based software that uses meta-content [3].

2. A SAFE RTS FOR WEB SERVICES

In principle, we follow the three main steps of the safe RTS technique: (1) constructing CFGs of old and new program; (2) identifying dangerous edges by comparing the corresponding CFGs; (3) selecting test cases that need to be rerun. CFGs are generated from actual programs in any language or extracted from designs, which can be used as a common representation mechanism among Web services.

2.1 Constructing Global Control-flow Graphs

At each service, a CFG is generated for every operation. Every CFG is identified by its corresponding operation name and the service's URI. In terms of granularity level of the CFGs, on one hand, we want to drill down to the statement level. In a statement-level CFG, each statement corresponds to a node. To support future comparison, every node records a hash code of its corresponding statement. With such a detailed CFG, we will be able to precisely predict the impact scope of a given code modification. On the other hand, we want to be able to ignore any unnecessary details. In every modern Web service implementation, service implementations run in a framework such as the J2EE server or the .Net framework. It is safe for us to treat the code provided or generated by the frameworks as unchanged libraries, and omit them in our analysis. Thus, the CFG of each operation in each service will be generated from only those methods (functions) which are actually implementing the operation. The CFGs of some services can be abstracted to a higher level such as the block level or the method (function) level. This node records a hash code of the entire unit of the code. Every CFG node carries four pieces of information represented by the self-interpreted variable names: (service_operation_ID, granularity_level, hash_code, is_changed, is_call_node).

We limit our approach to static service composition only. That is, every called operation is predetermined by the program without involving service discovery and lookup. We have to analyze the code and recognize the operations and their belonging services that the subject program calls. (The subject program can be either a client or a composite service.) In our experiments, we particularly studied the Axis Web service proxy in which the service locator object holds the service URI upon instantiation. With the reference of the WSDL document of the Web service corresponding to the URI, we can scan the subject program and find each service-call statement. For each remote service call, we create a special node in the CFG called "call node". Every call node records the operation name and the service URI of the call.

If a CFG has no call node, it is a terminal graph because it is ready to support RTS processing. However, if a CFG contains a call node, then this CFG cannot directly support RTS processing. Thus we call such a CFG a non-terminal graph. Once all the CFGs are in place, we can convert every non-terminal CFG to a terminal one by inserting the corresponding CFG into each call node. Since the inserted CFG may contain call nodes, the conversion process is generally recursive. In updating and transmitting CFGs, we impose a rule that allows any CFG holder to deliver terminal CFGs only; this rule eliminates any unnecessary communication.

2.2 Identifying Dangerous Edges

When the code of an operation is modified, a new CFG is created, which is then compared to the old CFG by performing a dual-traversal of both CFGs. Such a comparison determines which parts of the graph are different. The differences can be either structural (topological) change or a modification of the contents of the corresponding nodes. Detecting the structural difference is straightforward because the dual-traversal comparison is devised for this purpose. The content difference is detected by comparing the hash code stored in both of the CFG nodes. Once the changed nodes are marked, the downstream edges in the CFG are marked as dangerous. When applying this approach to Web service system, we have to consider two special situations among others: (1) the modification is made in a different (remote) site; (2) multiple changes happen concurrently.

Case (1) is caused by the distributed nature of Web services. A modification of an operation, *M*, in a service can affect other services' operations that directly or indirectly call *M*. To find the set of these potentially impacted remote operations and their hosting services, we use a *call graph*, in which each node (vertex) represents an operation or a client program; each arc represents a *call* relationship: the operation (or the client program) represented by the source node calls (invokes) the operation represented by the pointed node. Every node is associated with a box that represents a Web service. The "service boxes" merely indicate the locality of the operations. An example of such a call graph was shown in Fig. 1. Obviously, if operation *x* calls operation *y*, then a change in *y* may affect *x*. Let *P* be a node, and *p* be the operation corresponding to *P*. In general, starting from *P*, all the reversely reachable nodes compose of the inverse closure of *P* on the call graph. The operations corresponding to the nodes in the inverse closure are all the possibly affected operations due a change in *p*. They should be tested. Thus, the services hosting these operations should carry out a round of RTS process.

Case (2) is interesting and challenging. Autonomous services can modify their operations concurrently. The RTS will be carried out on each site separately. If there are failures, we wish to be able to precisely determine which change is causing the trouble. However, two changes in a system may conflict in such a way that there is no way to determine which change caused the failure upon the tests. We need to find a good order to select the test cases and to run the cases, if concurrent running is not allowed. Here, we use the call graph to help us analyze.

If two changes happen in two operations that are mutually independent, that is, none of these two operations is reachable from the other operation in the call graph. Obviously, these two changes can be tested concurrently. We will be able to determine which change cause the fault and act appropriately.

If the two changes occur on the same path, that is, one changed operation is reachable from another changed operation in the call graph the two changes conflict. We use a rollback mechanism to handle conflict. Cascading rollbacks, which may occur in this scenario, must be avoided by forcing changes in a call graph path to be applied in the order of "downstream changes first". Once a failure happens, no upstream change should be attempted before the troubled unit is fixed.

2.3 Selecting Test Cases

The result of the second step is a set of dangerous edges. Using a table that recording the coverage relationships between the test cases and the edges, selecting test cases is straightforward given the set of dangerous edges. This is performed in the exact same manner as described by [1].

3. AUTOMATING THE FRAMEWORK

A CFG is generated, integrated, stored and maintained by an RTS agent at every participating service site. A straightforward way to synchronize the RTS processes at multiple sites is to establish a central Web service that creates and maintains the global call graph. This service would have three operations: request_change (op_ID), report_pass(op_ID), withdraw_change (op_ID). An alternative distributed solution can be implemented using the event-subscription model. Once every agent completes the subscriptions for every remote operation, the RTS agents start periodical CFG updating following a token-passing algorithm, the goal of which is to enforce the "downstream changes first" rule in CFG updates and RTS processes among all the participating service sites.

4. REFERENCES

[1] G. Rothermel and M.J. Harrold, "A safe, efficient regression test selection technique", ACM Trans. Softw. Eng. Methodol. Vol. 6, No. 2, 173-210, April 1997.

[2] W. T. Tsai, et al, "Extending WSDL to facilitate Web services testing", Proceedings of 7th IEEE HASE, pp. 171-172, 2002.

[3] A. Orso, et al, "Using component metacontent to support the regression testing of component-based software", Proceedings. IEEE ICSM, pp 716-725, 2001