WebRTS: A Dynamic Regression Test Selection Tool for Java Web Applications

1st Zhenyue Long
GuangDong Power Grid, GuangDong
State Key Laboratory of Computer Science
Beijing, China

2nd Zeliu Ao
University of Chinese Academy of Sciences, Beijing, China

3rd Guoquan Wu*
State Key Laboratory of Computer Science
University of Chinese Academy of Sciences
Nanjing Institute of Software Technology, ISCAS

4th Wei Chen
State Key Laboratory of Computer Science, ISCAS
University of Chinese Academy of Sciences
Nanjing Institute of Software Technology, ISCAS

5th Jun Wei
State Key Laboratory of Computer Science, ISCAS
University of Chinese Academy of Sciences
Nanjing Institute of Software Technology, ISCAS

***Abstract*—Regression testing is an expensive activity in software development. To speed it up, regression test selection (RTS) is a promising approach by selecting a subset of tests which are affected by code changes. Although there are lots of regression test selection tools, most of them aim to unit tests, require direct code dependency between tests and code under test, and cannot be applied to Web applications to select end-to-end web tests. This paper presents WebRTS, a dynamic RTS tool for regression testing of Web applications. By tracking the process of Http request and object construction in the server, WebRTS can collect accurate test dependencies for each test in isolation, and supports parallel regression testing of distributed Web application. The design of WebRTS is also flexible, and it can be combined with different web testing frameworks. The experimental results show that WebRTS is effective and can be used to select regression tests for Java Web applications. Video: https://youtu.be/OlAsvrX7HXc. Source code: https://gitlab.com/aozeliu18/webrts**

I. Introduction

Regression testing is a well-established software testing technique, which is commonly used in industry to ensure that incremental updates to software do not break existing functionality. Regression Test Selection (RTS) [11] techniques optimize regression testing by skipping tests that are not impacted by the changes introduced since the last test execution. To this end, test dependencies are automatically tracked between executions. Dependency tracking can be done at different level of granularity– statement, method, file, or module – leading to different trade-offs. Tests whose dependencies did not change since the last test execution can safely be skipped. An RTS technique is safe if it does not miss to select any affected tests and precise if it selects only affected tests.

RTS can collect dependencies statically or dynamically. Recently, both Ekstazi (a dynamic RTS technique) [3] and STARTS (a static RTS technique) [8] show that performing RTS at the class level gives better speedup than performing RTS at the method level. Ekstazi [3] instruments the test code and the code under test to collect class-level test dependencies while running tests. Practitioners have adopted Ekstazi [3] and integrated it in the build systems of some open-source projects (e.g., Apache Camel).

Despite recent progress made in the RTS techniques, most of them aim to unit testing, and cannot be applied to end-to-end (E2E) Web application testing directly for the following reasons: 1) In unit testing, each test has direct dependency with the code under test. Web application tests, on the other hand, run in a separate environment from System Under Test (SUT), and communicate with SUT through Http messages. Such dependencies cannot be obtained directly; 2) To improve efficiency, it is common to run tests in parallel during the regression testing. For unit testing, the test code and the code under test run in the same process. As the test environments are isolated from each other for different tests, it is easy to support parallel test execution without affecting test dependencies collection. However, in the web domain, all tests share the same SUT. It is non-trivial to collect correct test dependencies for each test when running them in parallel; 3) For Java-based Web applications, some popular frameworks (e.g., Spring) support to load objects eagerly during the startup of SUT, or initialize them lazily until they are firstly used. These objects will be shared across different user/test requests. Applying existing class-level dependencies collection technique (proposed by Ekstazi for unit testing) directly on web tests selection may not be safe, as it assumes the codes under test are independent from each other for different unit tests, which does not hold for E2E web application testing. To improve safety, one possible way is to add all objects created during the startup of SUT as the dependencies of each test case. However, this approach will make RTS techniques select excessive tests, reducing their efficiency.

To address above challenges, in this paper, we designed WebRTS, a novel dynamic RTS tool for Web applications. WebRTS has four unique features: 1) it supports to track test dependencies when running tests in parallel; 2) it can be applied for distributed Web applications, in which the modules are deployed in distributed environment; 3) it considers object

* Corresponding Author. Email: gqwu@otcaix.iscas.ac.cn

2576-3148/20/$31.00 ©2020 IEEE
DOI 10.1109/ICSME46990.2020.00102

sharing in SUT between different tests, and can accurately select web tests by tracking the construction process of the objects; 4) different from existing dynamic RTS techniques which are tightly coupled with the testing framework (e.g., jUnit), WebRTS can be easily integrated with different web testing frameworks (e.g., Selenium, Robotium).

In our current implementation, WebRTS aims to regression tests selection for Java Web applications as they are widely deployed in the enterprise. However, the basic idea of WebRTS can be applied to Web applications implemented in other languages.

II. WebRTS

In this section, we introduce the system architecture and the implementation of main components in WebRTS.

A. System Overview

In order to adapt to different web testing frameworks, WebRTS is designed to be loosely coupled with web testing framework. Figure 1 shows its overall architecture, which mainly consists of four components: *Agent*, *Collector*, *Selector* and *CheckSum*.

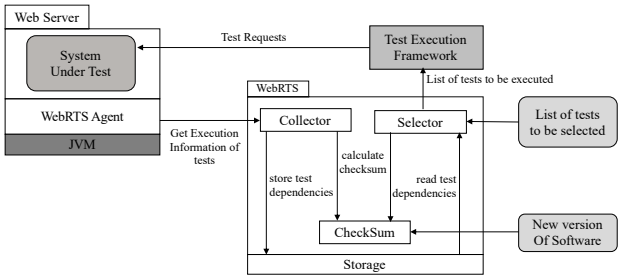

Fig. 1. Overall Architecture of WebRTS

A typical RTS has three phases: the analysis phase selects the tests to run in the current version, the execution phase runs selected tests, and the collection phase collects test dependencies for the next revision.

In practice, as Web applications are usually deployed on distributed nodes, WebRTS *Agent* needs to be installed on each server node, and it will record the execution information of http request in the sever during the execution phase. After all tests are completed, *Collector* will pull the recorded execution information from each node, and compute the dependencies for each test. Similar to Ekstazi [3], WebRTS computes test dependencies at the class/file level granularity. It invokes *CheckSum* to compute the checksum of each dependent file, and then save them to the dependency storage. When a new version of SUT is deployed, WebRTS enters into the analysis phase, and it invokes *Selector* to select tests affected by code changes based on the recorded dependencies in the old version. In the following, we first introduce the designed test dependency format for regression web tests selection before depicting each component in detail.

B. Test Dependency Format

WebRTS computes test dependencies at the file level. However, different from existing RTS techniques for unit testing, which collects test dependencies for each test directly, in WebRTS, web test runs in a separate environment from SUT, and interacts with backend server through Http requests. Therefore, we designed two-layer test dependency format for each web test (see Figure 2), in which each test depends on a list of triggered http requests, and each request maintains a list of class/file level dependencies.

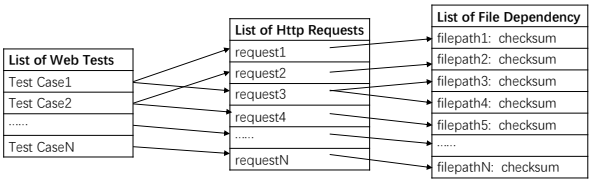

Fig. 2. Dependency format in WebRTS

The designed dependency structure also brings the following two advantages: 1) it explains why code changes of a file impact a web test by specifying the associated http request; 2) it can avoid repeatedly collect test dependencies for the same http request (e.g., the login request) which exists in multiple test cases, and thus can improve the overall efficiency of WebRTS. An example of recorded test dependency is shown in Figure 3.

Fig. 3. An example of recorded test dependency

C. Agent

The *Agent* will track the execution of HTTP request (sent by the web test) in the server. Figure 4 shows its overall architecture, and it is further divided into five modules: *Interceptor*, *Constructor Monitor*, *Http Request Processor*, *Http Call-chain Tracer* and *Execution Info. Container*. Note that, for E2E web testing, some objects in SUT will be shared across different tests. To avoid missing dependencies caused by objects sharing, all methods in the classes will be instrumented, and *Agent* will further track the construction process of all objects during the process of http request.

1) Interceptor: It is implemented based on JVM-SANDBOX [7], which is a dynamic instrumentation tool based on JVMTI and ASM technology. *Interceptor* uses JVM-SANDBOX to dynamically instrument SUT to collect execution information. *Interceptor* instruments the following code

points for each class: 1) *start* and *end* of a constructor; 2) *start* and *end* of the entry method of http request processing; 3) start of the remaining normal methods. Besides, we also instrument standard Java library methods that may open a file (e.g., FileInputStream) to record accessed files (e.g., javascript/css file) during the process of a test request.

2) Constructor Monitor: For E2E web application testing, some objects in SUT will be shared across different tests. To record such dependencies, this module is responsible for tracking the construction process of each object. As it is common to use thread model to handle each request in Java Web application, this module initializes a hashmap structure (called *objDepdency*) in the ThreadLocal context to maintain the dependent classes for each object construction, where *key* is the reference of object, and *value* is a set which stores all dependent classes (the class names) to construct this object.

When the *start* event of a constructor is received, *Constructor Monitor* starts to track the construction process of this object by initializing a set, which saves the class of this object and all its superclasses. Note that, it is possible that a new internal object will be constructed during the construction of one object. This module uses a stack structure (called *RecordStack*) to maintain the nested relationship in the object construction. When the *end* event of internal object's constructor is received, its dependency set will be removed from the top of the *RecordStack* and added into *objDepdency*. Note that, this dependency set (of internal object) will also be merged into the corresponding one of its outer object, as the outer object depends on its internal object.

For a normal method invocation (including file access), it will be handled by both *Constructor Monitor* and *Http Request Processor*. *Constructor Monitor* will check whether there exists any object being constructed (by checking whether *RecordStack* is empty). If true, this event will be discarded. Otherwise, *Constructor Monitor* will determine the object that invokes this method, try to get its dependencies by querying *objDepdency* and merge it with the dependencies of the top element in the *RecordStack*.

3) Http Request Processor: It is responsible for tracking all involved objects that handle a http request by monitoring all methods invocation. After entering the entry method of http request, this module will initialize a set (called *reqObjs*) in the ThreadLocal context, which records all used objects by monitoring normal method invocation (that handle this request). At the end of the entry method (of http request processing), this module will save *reqObjs* and *objDepdency* to the module Execution Info. Container.

In order to facilitate the computation of test dependencies, WebRTS needs to distinguish which test that each HTTP request belongs to. To achieve this, we embed a proxy (currently we use BrowserMob-Proxy [2]) into each test case, which will intercept http request to insert a test ID in the header before the request is sent to the server.

4) Http Call-chain Tracer: For distributed web application, one server may need to send a http request to another one during the process of a test request. To correlate the distributed calls belonging to the same test request, this module leverages existing distributed tracing tool to transmit transaction ID (inserted into the request header) along the call chain. Currently, we choose pinpoint [9], but other tools (e.g., ZipKin, Jaeger) are also applicable.

5) Execution Info. Container: After receiving the *pull* command from the *Collector*, this module will traverse *reqObjs*, and compute the dependent classes by querying *objDepdency* merged from different threads. Note that, it is possible that shared object also exists in the *objDepdency*, and in this case, WebRTS will query *objDepdency* recursively to retrieve dependent classes.

6) Optimization: To reduce the overhead caused by dependency tracking, for each test request sent from the testing framework, WebRTS will first check whether this request has been tracked in other tests. If true, it will not track this request. To implement this, WebRTS modifies the transaction ID to append a special string (e.g., -notrack). Then each *Request Processor* in the call chain will first check the transaction ID, and if it contains a special string, it will not initialize *reqObjs* to track this request.

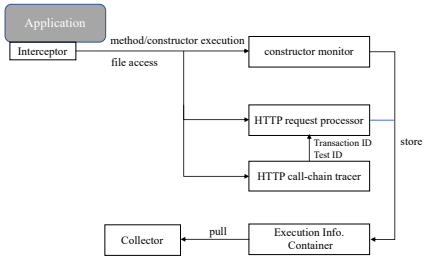

Fig. 4. Architecture of Agent

D. Collector

The Collector component is responsible to compose tracking information retrieved from *Agents* which are deployed in multiple servers after all tests are completed. It first correlates the dependencies collected from multiple nodes belonging to the same request according to transaction ID, and then converts the recorded classes into file dependencies. After that, it calculates a checksum for each class file, and saves the calculated dependencies into the storage.

E. Selector

This component is responsible for checking whether the checksum of each file in the revision has changed compared with corresponding one in the old version. If it is true, the dependent test is selected. The computation of checksum for each file is similar to the technique used in Ekstazi [3].

Note that, by interpreting some common configuration files (e.g., applicationContext.xml in Spring, mapper.xml in MyBatis) to associate configuration items with the class files, WebRTS also supports to select tests caused by configuration changes. For space limitation, we do not describe it in detail.

III. Evaluation

WebRTS is implemented using Java language. To evaluate its effectiveness, we chose two widely RTS metrics: selected

TABLE I
DETAILS OF THE SUBJECTS USED IN OUR EMPIRICAL EVALUATION.

Subject	#File	#KLoc	#TC [avg]	#Ver.	Time (minutes)		WebRTS		WebRTS*	
					avg.	total	e%	t%	e%	t%
JPetStore	240	13.5	10	4	2.0	8	20	25	60	64
Agilefant	389	26.3	65	8	8.7	70	17	23	33	39
PAM (core)	>1000	>300	90	5	288	1,443	23	28	47	53
eSMM(core)	>10,000	>2,000	142	8	327	2,613	18	25	43	45

tests and testing time, and compared WebRTS with RetestAll strategy (which runs all tests without WebRTS integration). To evaluate the proposed object construction tracking technique, we also implemented a modified version of WebRTS (called WebRTS* in the following), which instruments the same code points as Ekstazi (e.g., the start of a constructor) and adds all constructed objects during the startup of SUT to the dependencies of each test (similar to Ekstazi's method-level selection granularity by appending dependencies collected during constructor invocation and setUp methods to the dependencies of each test method).

For the evaluation, we selected 4 subjects, in which *JPetStore* [6] and *Agilefant* [1] are two open source Java Web applications, and *eSMM* and *PAM* are two enterprise Web applications from our industrial partner (a large power grid company in China). The basic information of these projects can be found in Table 1. For *JPetStore* and *Agilefant*, they are deployed on a 4-core 1.6GHz intel i7 CPU with 16GB memory running Ubuntu Linux 18.04 LTS, and for eSMM and PAM, they are deployed on a clustering environment consisting of 23 nodes in our industrial partner. For *JPetStore*, it contains 10 E2E tests, and we selected 4 iterations from the commits history in which the source code changes, and for *Agilefant*, we generated 65 tests based on the test scenarios described in the work [5], and selected 8 releases from v3.3.2 to v3.5.3. For *eSMM* and *PAM*, we selected the revisions released in the last 2 months. Note that, in the experiment, we only consider core tests for eSMM and PAM, as running all tests (>700 for eSMM and >500 for PAM) will spend more than 24 hours using single machine. For *Time* column, *Avg./Total* represent the average/total execution time of all tests across selected revisions for each subject. The last 4 columns in Table 1 show the percentage of selected tests (e%) and the execution time normalized by time of RetestAll (t%) for WebRTS and WebRTS*, respectively. It can be seen that WebRTS is effective in reducing selected tests and testing time during the regression. For example, for eSMM subject, WebRTS selects on average 18% of tests, and the time that WebRTS takes is 25% of RetestAll strategy. For WebRTS*, it selects more tests than WebRTS as more dependent classes (the class files of initialized objects during the startup of SUT) are added to the dependencies of each test, leading to more tests being selected (if any of these class files change), and more testing time being spent.

IV. Related Work

Regression test selection has been studied for several decades [11] [3] [13] [8]. Early RTS techniques select tests based on fine-grained block level [11] or method level [10] granularity, which may incur large overhead. Recently, Gligoric et al. [3] proposed file level RTS for Java projects, which outperforms method-level RTS in terms of end-to-end testing time. Zhang [13] presented hybrid RTS, which combines method and file level granularity to increase precision of RTS.

For regression testing of Web application, Xu et al. [12] proposed an RTS technique based on slicing, which models Web applications as system dependency graph and selects tests that execute the potentially affected web elements. Huang et al. [4] proposed RTS for J2EE application, which analyzes both code files and configuration files. TestSage [14] is a dynamic, method level RTS technique for large-scale Web services. All these work do not consider object sharing across tests and they are also not friendly to parallel testing.

V. Conclusion

This paper presents WebRTS, a novel dynamic regression tests selection tool for Java Web application. By tracking the process of the objects construction and HTTP requests call chain in the server, it can compute accurate test dependencies, and support regression testing of distributed Web application. One limitation of the current tool is that it considers JavaScript files running in the browser as resource files, and if the file changes, all tests that use it will be selected. We plan to support fine-grained dependencies collection for JavaScript files in the future.

VI. Acknowledgments

This work is supported by GuangDong Power Grid Company Limited under Project 037800KK52180009.

References

[1] Agilefant. https://github.com/agilefant/agilefant.
[2] browsermob proxy. https://github.com/lightbody/browsermob-proxy.
[3] M. Gligoric, L. Eloussi, and D. Marinov. Practical regression test selection with dynamic file dependencies. In *ISSTA*, 2015.
[4] S. Huang, Z. J. Li, J. Zhu, Y. Xiao, and W. Wang. A novel approach to regression test selection for j2ee applications. In *ICSM*. IEEE, 2011.
[5] W. Jin, T. Liu, Y. Cai, R. Kazman, R. Mo, and Q. Zheng. Service candidate identification from monolithic systems based on execution traces. *IEEE TSE*, 2019.
[6] JPetStore. https://github.com/mybatis/jpetstore-6.
[7] JVM-Sandbox. https://github.com/alibaba/jvm-sandbox.
[8] O. Legunsen, A. Shi, and D. Marinov. Starts: Static regression test selection. In *ASE*. IEEE, 2017.
[9] Pinpoint. https://github.com/naver/pinpoint.
[10] X. Ren, F. Shah, F. Tip, B. G. Ryder, and O. Chesley. Chianti: a tool for change impact analysis of java programs. In *OOPSLA*, 2004.
[11] G. Rothermel and M. J. Harrold. A safe, efficient regression test selection technique. *TOSEM*, 6(2), 1997.
[12] L. Xu, B. Xu, Z. Chen, J. Jiang, and H. Chen. Regression testing for web applications based on slicing. In *COMPAC*. IEEE, 2003.
[13] L. Zhang. Hybrid regression test selection. In *ICSE*. IEEE, 2018.
[14] H. Zhong, L. Zhang, and S. Khurshid. Testsage: Regression test selection for large-scale web service testing. In *ICST*. IEEE, 2019.