A Target-driven Framework for Distributed Software Test Automation

Jian-hua Gu
School of Computer
Northwestern Polytechnical University
Xi'an, P.R.China
e-mail: gujh@nwpu.edu.cn

Xing-she Zhou
School of Computer
Northwestern Polytechnical University
Xi'an, P.R.China
e-mail: zhouxs@nwpu.edu.cn

***Abstract*—This paper presents a four-layer framework for distributed software test automation, which is composed of test component layer, test component management layer, test scheduling layer and test domain layer. According to the target of testing, the problem space of software test divide into several feature tests, and each feature test is composed of a group of test cases. Employing the thought of software component, we create each test case as a component and then construct the test component library for a special application's test. During testing, with component management tools, the test engineer can select the proper test components manually or automatically according to the testing target. Then, these testing components are scheduled by the Harness automatically. Finally, the testing result reports are generated automatically, thus the test for a certain testing target is completed. With its good flexibility and scalability, this framework has been successfully applied to the system test of a distributed software.**

Keywords-Target-driven; Software Test Automation; Component; Test Case; Test Schedule

I. INTRODUCTION

The software testing plays the key role in software lifecycle and the improvement of software testing quality and efficiency plays an important role in shortening lifecycle of software development and improving the quality of software development. The complexity of the software makes the software testing more and more difficult. The use of automated test technique is essential for achieving high quality software and for shortening software lifecycle.

Both academy and industry field have done some research on the software test automation and have developed a several tools， for example Rational TestStudio[7]， WinRunner software[8] and Parasoft Software Testing Tools[9]. [4] designed a framework for web-based software testing that focuses on scalability and evolving the test suite automatically as the application's operational profile changes. [5] described the experience of creating of the automated test system as well as using it for testing embedded real time on-board software developed for the European Space Missions COROT. [6] developed a web application model and set of tools for the evaluation and automation of testing web applications. [10] presented an automatic conformance testing tool with timing constraints from a formal specification of web services composition. [11] focused on automatic test data generation from a stand-alone executable.

Because of the diversity of software function, common automatic testing tools usually are dedicated to the special software, in lack of general framework. After analysis of the software test process, this paper presents a target-driven four-layer framework for software test automation based on component. With its good flexibility and scalability, this framework has been successfully applied to the system test of a distributed software.

The rest of paper is organized as follows. Section II presents detail of four-layer framework of distributed software test automation. Section III and Section IV presents the analysis and design, and management of test component respectively according to the testing target. Section V and section VI shows how *Harness* to schedule the test components and to give the testing report automatically. Finally, Section VII concludes the paper.

II. THE FRAMEWORK

In general, software test includes following actions[3]: presentation of testing condition, design of test case, execution of testing case and analysis of testing result. Among these four actions, the former two actions focus on the intelligent actions which determine the quality of test case and they take place only once in a whole view. Whereas the later two actions usually need to be run many times, so they are suitable for automation. So we focus on the automatic testing technology of execution and analysis.

This paper presents a target-driven four-layer framework of software test automation (see Figure 1), which includes test component layer, test component management layer, test scheduling layer and test domain layer. The principle of the framework is that we can divide the problem space of software test into several feature tests according to the target of test, and each feature test is composed of a group of test cases. Employing the thought of software component, we create each test case as a component, thus the test component library for the special software test can be created. During testing, with component management tools, the testing engineers can select the proper test components manually or automatically according to the testing target. These components may be grouped into several categories for special test features. Then, the selected testing components are scheduled by the test scheduler, the *Harness*, automatically. Finally, the report of testing results are produced automatically, thus the test for a certain testing domain and its target is completed.

When testing engineers plan to test the software for a special test target, for instance the regression test, they usually make out a testing plan, the blueprint, according to the target. At the same time, they can use components management tool to select some proper components (the test case) from the component library and to put these components into the component pool. They may configure the components in component pool if necessary. *Harness* and *Logscanner* will schedule the components in component pool automatically and produce the testing report (See Figure.2).

III. TEST COMPONENTS

A. *Principle of Component Design*

The first step to build the component is to analyze, abstract and model the testing problem space according to software under test and test target. To build a test component should follow these principles:

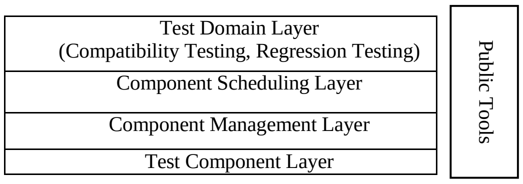
Test Domain Layer
(Compatibility Testing, Regression Testing)
Component Scheduling Layer
Component Management Layer
Test Component Layer
Public Tools

Figure 1 Framework of Automatic Testing

1) Independent: The test component should be independent to each other, so one test component cannot influence the execution of other components whether or not a test component executes successfully. Thus, the sequence of component execution in schedule layer cannot be considered.

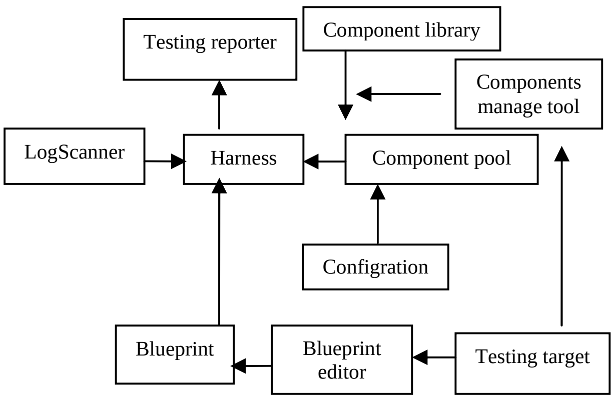

Figure.2 Detail View of Framework

2) Cohesion: The test engineers need not know the inside of component in detail and just know the interface and function of test component.

3) Configuration: The purpose of test component configuration is to make the component fit to different test requirements and to enhance the reusability of component.

4) Easily understanding and using: The function of component should be easily understanding with specification. The special skill of test engineers is not necessary.

5) Portability: The tested object always run on different operating system platforms, so test component need a good portability.

B. *Model of Test Components*

By analysis, we abstract a testing component with a five tuples.

aTestComponent = <Name, TestEntity, Interface, Output, Descriptions>

It denotes respectively name, entity, interface, output and description of a component.

1) Name: It is the unique identification of component, adapting hierarchy name, for instance: test category name, test group name, test function name, test number, and so on.

2) Entity: The entity consists of:

```
TestEntity
  {  attributes;           // member variables
     constraints;           // limited condition
     init();               // constructor
     pre_condition();      // prepare processing condition
     processing();         // execution body
     post_processing();    // post process
     config();             // configuration
  }
```

The entity of a test component is the main execution part. It is composed of two sections: data section and the processing section. The *attributes* in data section declare some data structure used in component. The *constraints* in data section declare some limited information to component, for example, time constraints. The testing result cannot be accepted if execution of component is time-out even if the component runs normally. The processing section is composed of three steps. The *init()* sets up the environment and *pre_condition()* makes preparations for *processing()* if necessary; the *processing()* executes the test case; the *post-processing()* has two tasks: analyzing the result and cleaning the execution environment. The *config()* completes the component configuration for adapting different testing environments .

3) Interfaces: The users can use a component by interfaces. In fact, interfaces can divide into two parts: the interface of testing function and the interface of configuration. The former is used to accomplish the testing function and the later is used to change the attributes of components. The interface encapsulates the implementation of components and makes the component more independent. With the relationship among the component reduced, the component can interact with each other by interfaces and the implementation of a component cannot interfere with that of other component. On the other side, the component can interact with outside by its interface, so how to implement the test component isn't influential to the way to invoke a component as long as there is no change on its interface.

4) Output: The result of testing.

5) Description: Some description about test component, including the testing goal, function, interface, usage, expected result and so on.

IV. Test Component Management

In order to manage component more conveniently, a test component library is constructed. The component library is a collection of test components, which are constructed according to certain semantic and organizing style. It is the important resource and constituent of accomplishing automated software testing. Different test component library can be constructed according to different test purpose and phase.

A. Component Library Management

Test component library management provides following functions:

1) Obtaining and skimming component

User can obtain components he wants by searching the library by component name, classification, function, interface pattern, test purpose and so on. Then the user examines the component for ensuring whether it meets test requirement. The testing component can be fetched from the library and be put into the test pool when it meets requirement. User can also create a set of rules according to test purpose and obtains corresponding components automatically by obtaining component software.

2) Component configuration

If user finds the selected component does not meet his test requirement perfectly, he/she can modify the component's attribute through the configuration function.

3) Component maintenance

It includes these functions: adding, deleting and updating the component library, storing components according to classification, management the right of adding and deleting components.

B. Component library construction process

Test component library construction is a dynamic process. First, we abstract the test problem and earn a corresponding component. Then utilizing the component-modeling tool, we model the component entity, interface, and so on. These components can be added to component library but user cannot make use of them direct now. Second, using component management tool to pick-up the component in the rough, we complete component development task by corresponding development tool. Finally, if the component function test passed, the component can be published. If it exist some problems, tested component should be updated after modified.

V. Test Component Schedule

Execution of automated software testing is carried out by scheduler, the *harness*[1]. By test blueprint, scheduler carry out automatically the test component's loading and running, organize and report the test result, and recover from breakpoint if error exists.

The test blueprint is a file that describes the test plan. In order to enhance the flexibility of test engineer's operation, blueprint supports a group of running modes: executing a certain classification of test components, executing an appointed list of test components, executing all test components with the special test aim and so on.

After running one test component, some message of running states return to the scheduler and the scheduler creates report of this component according to the states at the end of test. In common, the return states are following four results: *OK*, *Failed*, *Unknown* and *Time-out*. The scheduler can send test report to user in many ways that user has defined in advance, such as screen output, files and Email.

Because the scheduler usually runs in the unattended condition for long time, it must have strong ability of dealing with exception and recovery from failure. No matter whether test components or scheduler self could occur problem, the scheduler should try its best to avoid stopping the execution and only record these exceptions in the Log file unless the problem is fatal. If the harness has to stop for fatal error, the scheduler should support recovery from the breakpoint. When running the scheduler again, user can restart the test process from the place where the last exception took place last time instead of from beginning.

Because some test components have real-time feature, the scheduler should deal with component's timeout. When running time of the test component is beyond its restraint time the scheduler can invoke time-out function.

VI. Test Result Verification

Test result verification is an important part of software test. In ordinary test, result verification is accomplished through test engineer's view. Because test result is complicated, automated test result verification is always an important research problem. We adopt two kinds of automated test result verification method:

A. Direct verification method

It means that verification is completed in test component directly, and there are following methods:

1) Directly compare verification method:

$$\mathbf{RESULT_{exp} = RESULT_{real}}$$

The test result is claimed correct only when the real output result is same as the expected one. The $\mathbf{RESULT_{exp}}$ and $\mathbf{RESULT_{real}}$ prefer to the expected result and the real result of test component respectively.

2) Keyword searching method:

$$\mathbf{Key(RESULT_{exp}) = Key(RESULT_{real})}$$

The test result is claimed correct only when the expected output result and the real one has some same keywords.

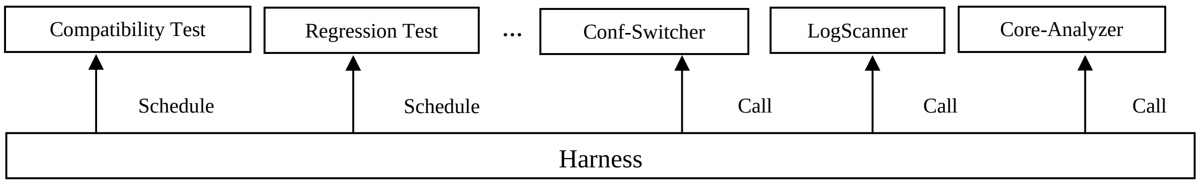

Figure.3 System Architecture of Auto_Tool

3) Limited value method:

$$|\, \mathbf{RESULT_{exp}} - \mathbf{RESULT_{real}} \,| <= \mathbf{THRESHOLD}$$

The test result is claimed correct only when the difference between of the expected output result and the real one is not higher than a certain limited value.

B. Indirect verification method

Because some distributed software is so complicated that we cannot succeed through direct verification method, we must adopt other indirect method to verify the test result. Log file is an important recorder of software system's running condition. User can aware of this system's running states by the log file. When exception happened, user can find corresponding information from the log file and then analyze the reason. Log file searching program, *LogScanner*, is used to analyze the log file produced by software under test. User not only can express own searching requirement flexibly and conveniently by logical expressions, but also can explain how to tell test engineers when corresponding message has been found out.

VII. CONCLUSIONS

The automated software testing framework presented in this paper has been applied initially in automated test of a distributed software LSF for Platform company. With guidance of this framework, we have developed a set of test components[2], which include test components for the distributed software configuration test, compatibility test and regression test. Other tools have also developed, for example, test component scheduling program *Harness*[1], log file searching program *LogScanner* and so on.

We developed an automated test system, called *Auto_Tool*[12]. Figure.3 is the illustration of the system architecture of *Auto_Tool*.

Harness governs the whole process of the automated test for the special testing target, for example the compatibility test, regression test and so on. It provides the strategies and methods to schedule and control automated test components in an automated software test. During test, if necessary, harness will start and control other test tools, such as *Logscanner*, *Conf-Switcher* and so on. Particularly, *Harness* provides programming interfaces for perl, c, c++, shell, java, which makes *Harness* easy to be used and expanded for test engineers.

Conf-Switcher automatically switches and changes the configuration of software under test according to the test target. At the same time, it checks whether the configuration of software under test is correct after switching or changing.

In order to verify the result of testing, *Harness* employs the tools *Logscanner* and/or *Core-Analyzer*. *Logscanner* monitors and controls the status and behavior of the tested system by analyzing a set of log files of tested software. When an exception or a piece of interested message is found, it deals with it or reports it to harness to deal with it. *Core-Analyzer* collects and analyzes the core files generated by tested software, which may be distributed on the different nodes.

ACKNOWLEDGMENT

Works partially funded by National High-tech R&D Program of China (863 Program,) under Grand No.2009AA01Z142 and the Research Fund of Platform company.

REFERENCES

[1] Wu Xiaojun, Gu Jianhua, ect.. Research and Implementation of General Automated Testing Tool: Harness[J]. Journal of Northwest University(Natural Science Edition). 2000（31）， 64~68.

[2] Liu Liang ect. Research and Development of Automated Compatibility Test Cases for Distributed Software System [J]. Application Research of Computers. 2003，20（2）：113~116.

[3] Zheng Renjie. Software Testing Technology[M]. Beijing : Tsinghua University Press，1992.

[4] Sreedevi Sampath etc. Composing a Framework to Automate Testing of Operational Web-Based Software. Proceedings of the 20th IEEE International Conference on Software Maintenance (ICSM'04)

[5] Natalie Russ etc . Lessons learned: on-board software test automation using IBM rational test realtime . Second IEEE International Conference on Space Mission Challenges for Information Technology, 2006（SMC-IT 2006）, 17-20 July 2006

[6] G.A.Di Lucca etc. Testing web applications. In International Conference on Software Maintenance, 2002

[7] IBM Rational Suite TestStudio . http://www-306.ibm.com/software/cn/rational/products/teststudio/index.html

[8] HP WinRunner software . https://h10078.www1.hp.com/cda/hpms/display/main/hpms_content.jsp?zn=bto&cp=1-11-127-24%5E1074_4000_100__

[9] Parasoft software testing tools. http://www.parasoft.com/jsp/products.jsp

[10] Tien-Dung Cao, Patrick Felix,etc. WSOTF: An Automatic Testing Tool for Web Services Composition. 2010 Fifth International Conference on Internet and Web Applications and Services

[11] S´ebastien Bardin and Philippe Herrmann. OSMOSE: automatic structural testing of executables. SOFTWARE TESTING, VERIFICATION AND RELIABILITY,2011; 21:29–54

[12] Liang Liu,etc. Agent-Based Automated Compatibility Software Test for NLSF[J]. Proceedings of the second International Conference on Machine Leamhg and Cybernetics, Xi'an, 2-5 November 2003