Automated Testing Framework Considering Distributed Testing Environment*

Hong Seong Park, Jeong Seok Kang, Wook-Jae Jo, and Mi-sook Kim, Kangwon National University

***Abstract*— Since component based development has been introduced in robot software, not only individual components but also combined components and their application tests are required. As the volume of the tests is increased, more automation of the testing framework is demanded to reduce time and effort. A test has a test resource management step, a test script generation step, a test resource changing detecting step, and a test execution step. Additionally, these steps needs are run in different environments, i.e. the user interface is through the Web, test resources are in a server, and the execution location for the test is in a robot. This paper introduces an automated test framework which runs on distributed environments.**

I. INTRODUCTION

Component based development has been introduced in the robot software field more than two decades ago [1-4.] Single components can combine to create a larger component or an application. This combining is simply connecting an input interface to an output interface of the two components in the software component based system (SCBS). Only the interfaces of the component are exposed to the outside from the system. These interfaces need to be tested during component and system development. Because a system is divided into many components, to test the system, a lot of component tests are involved. The automation of testing can reduce time and cost.

Up to now, investigated test automation techniques have had three steps: test data generation, test case generation, and expected result generation. Each step is automated using additional information, but there is no automation technique into which all of these test procedures are integrated. For SCBS testing, there is a need to automate these three steps to achieve efficiency.

This paper introduce a testing framework in which all three steps of SCBS for robot development are automated. Section 2 shows previous testing frameworks with Section 3 showing the elements of the suggested framework. Section 4 provides example cases of the automated testing framework. Further research topics will be discussed in Section 5.

*Resrach supported by OPRoS Project.

Hong Seong Park is with Kangwon National University, Chuncheon, S. Korea (phone: 82-33-250-6346; fax: 82-02-2212-7217; e-mail: hspark@ kangwon.ac.kr).

Jeong Seok Kang was with Kangown National University. He is now with ybrain (e-mail: jskang@kangwon.ac.kr).

Wook-jae Jo is with Kangown National University. Chuncheon, S. Korea (e-mail: jwj0104@gmail.com).

Mi-sook Kim is with Kangwon National University, Chuncheon, S. Korea (e-mail: kim.misook3@gmail.com).

II. RESEARCHED TEST AUTOMATION FRAMEWORK

The electronic business XML (ebXML) test framework [5] is a testing tool for ebXML confirmation and integration testing. It has 6 steps: test planning, designing test requirements, designing test suites, confirming the testing definition, and test suites execution. However, the test domain is limited to e-commerce.

The TETware framework [6] includes test related management, and test and result reporting. Also, the framework provides an API between the test process and test code for a confirmation test, and a performance and loading test. It includes a test case controller, common test suites' structures, test cases' structures, and various programming interfaces for test case development. The main function is the test case and the test code; this framework does not provide the functions for distributed environment.

The Test Process Management System (TPMS) [7] follows the characteristics of a test management supporting tool as defined by the International Software Testing Qualifications Board [8] and the test management process defined by the International Organization for Standardization [9]. TPMS includes risk analysis and management, design test strategy, test case execution and management, fault tracing and management, test report generation, and test activity management. This framework is mainly about risk management and reporting for general software development; it lacks functions for a distributed environment.

The Software Test Automation Framework (STAF) [10] tests with the communication of STAF Proc., a daemon, for various distributed test environments. STAF provides internal and external services supported by multiple operating systems and computer programming languages, and it is reusable. Major STAF services are processes, file system, compression, and monitoring as internal services, and e-mail management, event management, and HTTP [11] management as external services. This framework tests in various distributed environments, but its test resource management and test case generation are manual.

This paper introduces the SCBS test automation framework considering a distributed test environment. This framework has a Test Automation Engine (TAE), Test Agent (TA), Configuration Management Servers (CoM Server). TAE manages test activities, test design, test relationships, test environments, and test execution. TA is allocated in various test environments, and connected to the central TAE by a network and to a CoM Server to share the resources for test execution.

III. MAJOR FUNCTIONS IN AUTOMATED TESTING FRAMEWORK

Important functions of automated testing framework are testing resource management, test script generation, testing functions in changed resources, and test execution in a distributed testing environment

A. *Test resource management*

TA is a unit of test resource management, and it has information of SCBS and information of the test. Fig. 1 shows the class diagram of test activity.

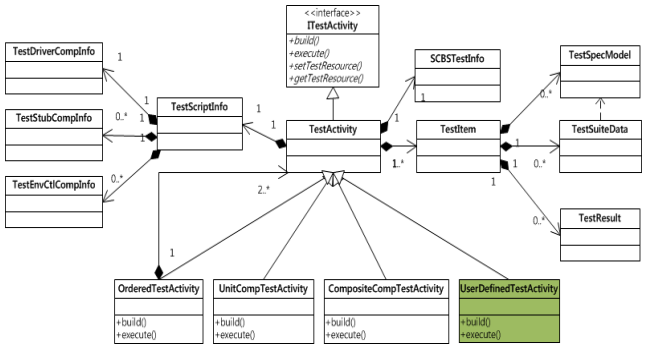

Figure 1. Class diagram of test activity

"*ITestActivity*" is an interface class for every test in common. The "*SCBTestInfo*" class is SCBS information of the testing target. The "*TestItem*" class has testing items of the test information. The "*TestScriptInfo*" class saves test script information in an automated test framework. The "*TestActivity*" class can be inherited to extend user-defined purpose.

B. *Test activity lifecycle*

A finite state machine is used to manage the 8 states of the test activity lifecycle. Fig. 2 shows the movement of the state. When a test activity is created, the state goes into the "Created" state. At the point a user requests a test execution, the state moves to "Waiting". If any test related resources are changed when the request arrives, then the state goes to "Changed" first, then moves to "Waiting". As it depends on the type of activity execution, when the execution time is reached, activity resources are deployed into TAs, and then the state changed to "Running." When an error occurs during deployment, the state goes to "Error." If the test was executed without errors, and all the test cases of the test activity are finished, then the state goes to "Success." If any test cases fail, the state moves to "Failure." If users choses to end the process during the test execution, then the state moves to "Stopped."

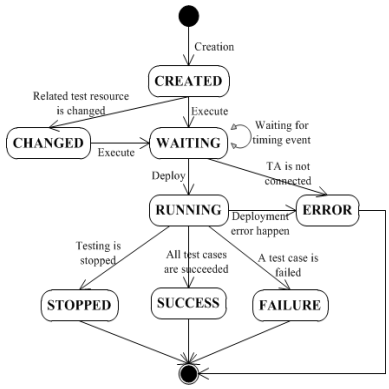

Figure 2. States of test activity

C. *Test activity execution types*

There are 4 types of test activity execution: immediate, reserved, periodic, and resource changing. It is determined in the Test Activity Scheduler (TAS) in Fig. 3.

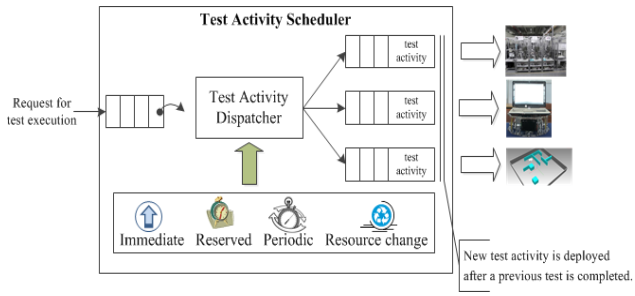

Figure 3. Scheduling of test activity

The TAS has queues on each execution type. When the execution command arrives from the users, TAS checks the type of test activity, then inserts the test activity into the queue for the type of test activity. The test activity stays until the previous test is finished, then the TA in the environment downloads the resources of the test activity, and tests it.

D. *Test Script Auto Generation*

The test script generator of the test automatic engine generates the source codes of the test driver component and test stub component. The test driver component manages the flow of the test and connects to the Provided Interface (an OPRoS component) of the SCBS. The test stub component connects to the Required Interface (another OPRoS component) of the SCBS. The test driver component (from OPRoS) reads the test cases and classifies them as Provided Interface (also from OPRoS). Then "TestControlInterface" is used to initialize the test stub component. Test cases are typed to call appropriate interfaces of SCBS, then the test results compared to the expected results and stored. The test stub component receives the test data, which are related to the Required Interface among the test cases, and is input into SCBS whenever the test cases are executed.

E. *Continuous test based on Resource-Change-Aware*

The test relationship management (TRM) module is executed to related test activity when test resources are changed. The test resource monitor, test resource analyzer, test

relationship manager compose the TRM. The test history of the test resources in the CoM Server is monitored periodically by the test resource monitor module. Then any changes of the test resource are sent to the test resource analyzer module. The module uses only changed information, excluding added or deleted information, to get the related test activity. The information is transferred into the test relationship manger module. The test relationship module has the execution manager re-run the changed test activity.

F. Automatic test execution in distributed test environment

The test automatic engine (TAE) runs tests by TAs allocated in various test environments and the CoM Server. There are the test agent registry step, test execution ready step, test execution step, and test result and reporting step.

In the test agent registry step, a test agent registers itself and its test environment information into the TAE. Then the test environment manager in the TAE stores this information into a database.

In the test execution ready step, a user requests an execution of test activity, then the execution manager checks whether any TA is connected in the test environment. If so, then the manager requests the execution to the activity scheduler and inserts the test activity into the test execution queue connected to the TA. The test activity distribution module requests the test agent to run the test activity in the execution queue. When the TA receives the test execution request, the test resource related to the requested test activity by the CoM Server is downloaded and finishes the readiness for the test execution.

In the test execution step, the test agent builds the test application in the downloaded activity. If it is a success, the test agent executes the application. When all the test cases are executed, the test application notifies the end of execution into the test activity executor.

In the test result and reporting step, the test agent uploads the result into the CoM Server, sends the result into the TAE. Finally, the TAE downloads the test result from the CoM Server and reports it to the user.

IV. RESULT OF IMPLEMENTATION OF TEST AUTOMATION FRAMEWORK

The TAE of the test automatic framework for various environments was implemented in Java [12], the database in MySQL [13], and the web-based user interface in Flex. The test agent module which runs tests in the various environments was implemented in Java, and the CoM Server, which shares test resources between test agents and test automatic engines, is implemented in SVN [14].

To verify the functionality of web-based test automation tool, OPRoS based robot software components were used: the Mobile Component, Distance Sensor Component, Control Component, and Statement Generation Component. Its structure is given in Fig. 4.

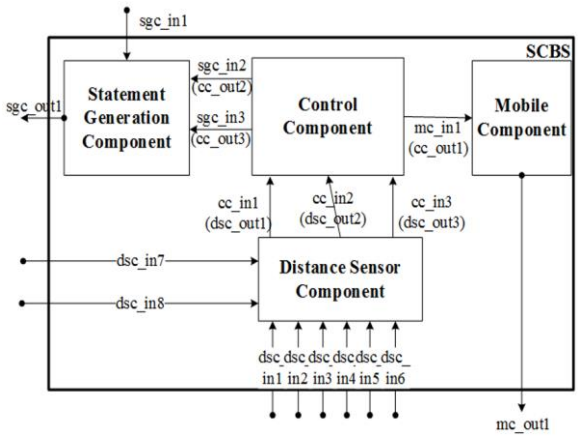

Figure 4. Components and interfaces for a test

This application is to print guide information while the mobile-base moves using a distance sensor to avoid obstacles. The Distance Sensor Component gets the distances through dsc_in1, des_in2, dsc_in3, dsc_in3, dsc_in4, dsc_in5, dsc_in6, dsc_in7, and dsc_in8 from the sensor hard ware. The Control Component calls the interfaces of the Distance Sensor Component through cc_in1, cc_in2, and cc_in3 to get the distances from any obstacles that the mobile base encounters. If the distance is less than a pre-determined distance then the Control Component calls the interfaces of the Mobile Component through mc_in1 to reduce speed of the wheel or rotate the direction of the mobile base. At the same time, the Control Component calls the interfaces of the Statement Generation Component through sgc_in2 and sgc_in3 to print the guide information. The Statement Generation Component prints the message when the entry is given through sgc_in1.

Fig. 4 shows there are 15 input interfaces to test in the application, 6 interior and 9 exterior. The test case was generated based on IOReACT [15]. For example, the interface sgc_in1 has a value of language types of a guide message. The test cases are a Korean type message and an English type message because users choose the language types for the interface test. Table 1 shows the number of test cases generated automatically for each interface.

TABLE 1 THE NUMBER OF TEST CASES OF INTERIOR/EXTERIOR INPUT

Internal / External Input	Number of Test Case	Internal / External Input	Number of Test Case
sgc_in1	2	dsc_in6	3
sgc_in2	4	dsc_in7	2
sgc_in3	5	dsc_in8	2
dsc_in1	4	cc_in1	4
dsc_in2	4	cc_in2	5
dsc_in3	4	cc_in3	4
dsc_in4	3	mc_in1	9
dsc_in5	3		

The test driver template code for test cases is generated as in Fig. 5.

```
}

bool ServiceMethodTDImpl::runTest(int testCaseID)
{
  if(testCaseID >= m_sizeOfTestCase){// 테스트 케이스ID가 테스트 케이스의 총 갯수보다 크거나 같으면 에러
      std::cout<<"<<SPT_LOG>>{TEST_ERROR}{TestCase ID is bigger than total number of TestCase}" <<
std::endl;
      return false;
  }
  if(m_methodTestCaseList == NULL){
      std::cout<<"<<SPT_LOG>>{TEST_ERROR}{MethodTestCaseList is NULL}" << std::endl;
      return false;
  }
  if(m_targetServiceObj == NULL){
      std::cout<<"<<SPT_LOG>>{TEST_ERROR}{Target Service Object is NULL}" << std::endl;
      return false;
  }


  /*************************************************************************************
   * TestCaseID로부터 해당하는 테스트 케이스를 읽고, 각 타입에 맞게 캐스팅 하는 루틴

  *************************************************************************************/

  // 테스트 케이스를 읽는다.
  MethodTestCase* testCase = m_methodTestCaseList->getMethodTestCase(testCaseID);


  // 각 테스트 케이스를 타입에 맞게 변환한다.



  /*************************************************************************************
  **
   * 테스트 대상 인터페이스를 호출하기 전에 수행되어야 하는 루틴

  *************************************************************************************
  /

  // TODO : 테스트 대상 메소드를 호출하기 전에 수행되어야 하는 작업을 구현하시오
```

Figure 5. Web based test drive template code

Fig. 6(a) shows choosing a test activity and compiling it over the Web. Fig. 6(b) shows that downloading the test activity resources from a SVN into a test agent in the robot and building these resources. Fig. 6(c) shows that the test is running while the robot is moving.

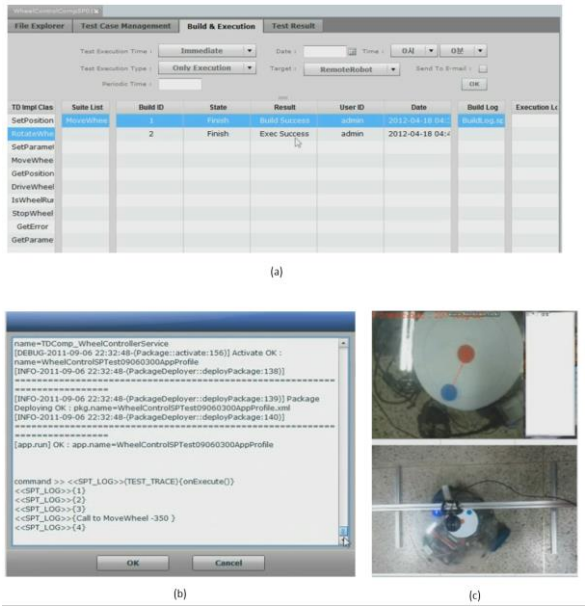

Figure 6. Distributed test environment for test execution

After all the test cases are executed in a physical robot, the result is in the table in Fig. 7(a), and its graphical test results are Fig. 7(b). The test shows that a test case of a 370 degree rotation shows the result false, which agrees with the expected result.

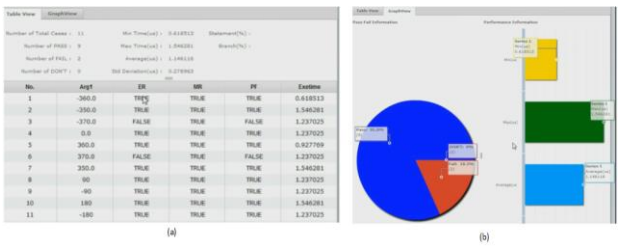

Figure 7. Result of the test in a real test environment

V. CONCLUSION

This test framework runs in various environments: a Web environment for receiving users' input and presenting the test result to users, a SVN server to maintain the test resources and test results, and a test agent in the robot. This framework delivers ease of use from automatic generation and distribution of testing driver and stub components to save time and effort. Among future research interests would be sharing various test environments in infrastructure as a service in a cloud.

REFERENCES

[1] S. Han, M.S. Kim, H.S. Park, "Open software platform for robotic services", in *IEEE Robot. Autom. Magaz.*, vol. 17 (1), pp.100-112, 2012.

[2] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foot, J. Leibs, R. Wheeler, A.Y. Ng, "ROS: an open-source Robot Operating System", in *ICRA Workshop on Open Source Software*, vol. 3, 2009.

[3] N. Ando, T. Suehiro, K. Kitagaki, T. Kotoku, W.-K. Yoon, "RT-middleware: distributed component middleware for RT(robot technology)", in *IEEE/RSJ International Conf. on Intelligent Robots and Systems*, pp. 3933-3938, 2005.

[4] H. Bruyninckx, "Open robot control software: the OROCOS project", in *IEEE International Conf. on Robotics and Automation*, vol.3, pp. 2523-2528, 2001.

[5] ebXML Test Framework Specification, Version 1.0, Technical Committee Specification, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=ebxml-iic.

[6] TETware, http://tetworks.opengroup.org/Products/tetware.htm.

[7] TPMS, http://www.sten.or.kr/index.php.

[8] ISTQB, https://www.istqb.org.com.

[9] ISO, http://www.iso.org.

[10] STAF, http://staf.sourceorge.net.

[11] R.T Field, J. Gettys, J. C. Mogul, H. F. Nielsen, L. Masinter, P. J. Leach, T. Bemers-Lee, "Hypertext Transfer Protocol-HTTP/1.1" IETF RFC 2616.

[12] Java, https://java.com.

[13] MySQL, https://mysql.com.

[14] SVN, https://subversion.apache.org

[15] Jeong Seok Kang, Hong Seong Park, "Input/output Relationship Based Adaptive Combinatorial Testing for a Software Component-based Robot System", in *Journal of Institute of Control*, vol.21(7), pp.699-708, 2015.