23

SITAF: Simulation-Based Interface Testing Automation Framework for Robot Software Component

Hong Seong Park and Jeong Seok Kang
Kangwon National University
South Korea

1. Introduction

Many researchers in robotics have proposed a Component-based Software Engineering (CBSE) approach to tackle problems in robot software development (Jawawi et al., 2007). Especially in the component-based robot system, the system quality depends on the quality of each component because any defective components will have bad effects on the system built with them. Thus, component interface test is critical for checking the correctness of the component's functionality. It is especially difficult to test robot software components because of the following two main problems.

First, the preparation of all hardware modules related to robot software and the configuration of a test environment is labor-intensive. Second, it is difficult to define or generate test cases for testing robot software components.

The simulation plays an important role in the process of robotic software development. The simulation allows testing of robot software components and experimentation with different configurations before they are deployed in real robots. Traditional simulation-based approaches (Hu, 2005, Martin & Emami, 2006, Michel, 2004) focus on architectures or methods (e.g., computer-based simulation, hardware-in-the-loop-simulation, and robot-in-the-loop- simulation), rather than testing. Many software engineering researchers (Buy et al., 1999, Bundell et al., 2000, Zamli et al., 2007, Momotko & Zalewska, 2004, Edwards, 2001) have investigated software component testing, but they have not considered simulation environments. Simulations can be used within a specification-based testing regime, which helps robot software developers define and apply effective test case. Note that the generation of test case is an important approach in the field of automated testing.

In this paper, we propose a Simulation-based Interface Testing Automation Framework (SITAF) for robot software components. SITAF automatically generates test cases by applying specification-based test techniques and considering simulation-dependent parameters. SITAF also performs the interface testing in distributed test environments by interacting with a simulation. SITAF controls test parameters during testing, which affect the behavior of a component under testing (CUT); examples of such parameters are simulation-dependent

parameters, input/output parameters of provided/required CUT interface. The main advantage of this technique is that it identifies errors caused by interactions between the CUT and the external environment.

The primary contribution of this paper is the integration of specification-based test into the simulation for automatic interface testing of robot software components.

The rest of this paper is organized as follows. Section 2 presents the SITAF architecture. The two main functions of SITAF are presented in Section 3. Section 4 discusses the evaluation of SITAF. Finally, we have some conclusions in Section 5.

2. SITAF architecture

The main aim of SITAF is to automate as much of test process for robot software component as is possible. To achieve this aim, the architecture of proposed framework consists of a Web-based Interface Testing Automation Engine Server (ITAES), a Test Build Agent (TBA), and a robot simulator and is shown in Fig. 1.

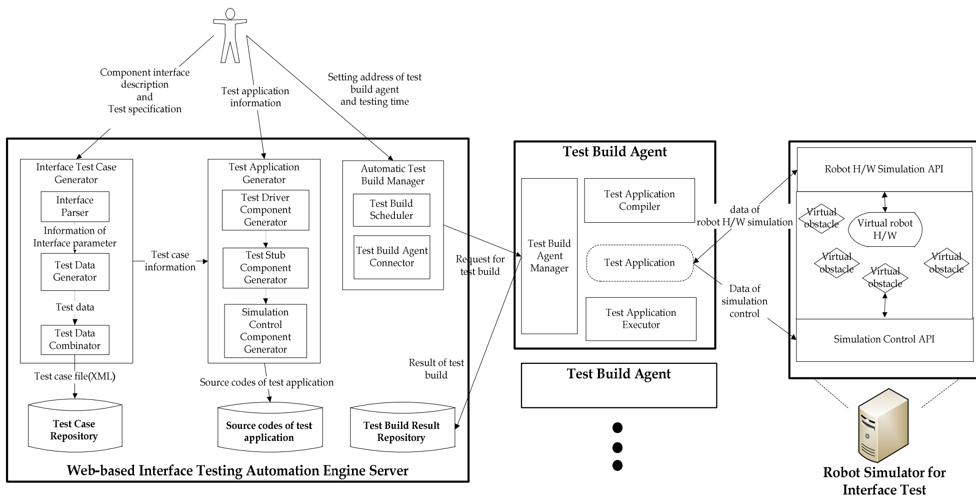


Fig. 1. Simulation-based Interface Testing Automation Framework Architecture

ITAES is the core of the framework to which the user is accessible via a Web service, and generates test cases for interface test of robot software components. And it also generates a test driver component, a test stub component, and a simulation control component, which are required for the testing, and links the generated components to each other. Furthermore ITAES manages test resources such as test cases, test applications, test results, and test logs in a unit of test activity which is a workspace for execution of test operations. The ITAES consists of three main modules of Interface Test Case Generator (ITCG), Interface Test Application Generator (ITAG), and Automatic Test Build Manager (ATBM).

The ITCG module automatically generates test cases by extended test schemes based on specification-based testing techniques such as equivalence partition (Ostrand & Balcer,

1998), boundary value analysis (Hoffman et al., 1999), random test (Ntafos, 1997, Zhu, 1997), and pairwise test (Williams, 2000). It receives the interface representation information in the form of Interface Definition Language (IDL) or eXtensible Markup Language (XML) and verifies the specification information for a CUT before test cases are automatically generated. The test cases are stored as XML files in a database. The user accesses a Web interface to modify test cases in the database and inputs the expected result values for each test case. The ITAG module generates the source code for the test application. The test application is composed of a test driver component, a test stub component, and a simulation control component. All source codes are shared with TBAs. The ATBM module connected with the distributed TBAs manages a test build which means compilation and execution of a test application. And it provides three types of the test time for the test build: immediate, reserved, and periodic.

An individual TBA can exist in different test environments, and communicates with the ATBM in ITAES. TBAs are in charge of automatic building of test application. The TBA contains three modules of Test Build Agent Manager (TBAM), Test Application Compiler (TAC), and Test Application Executor (TAE). The TBAM module manages a TBA and receives a test build request from the ATBM in ITAES, and then downloads the test application and test case files. The TAC module and the TAE module automatically compile and execute a test application. These modules upload the logs and the test results to ITAES after the completion of compilation and execution.

The robot simulator for interface test provides a simulation control API and a virtual robot hardware API. The simulation control API is used to control the virtual test environment in the simulation. The simulation control component in the test application dynamically modifies or controls a virtual test environment for each test case by using the simulation control API. The virtual robot hardware API is used to control virtual robot hardware or to receive data. If the test component is a hardware-related component, the component controls the virtual robot hardware or receives data using the API for simulation. The simulation control data is used to generate effective test cases.

3. Automatic interface test operations for robot software components

In this section, we describe two main functions of SITAF, which are the automatic generation of interface test case and the automatic execution of interface test by simulation.

3.1 Automatic generation of interface test cases

Specification-based test techniques are applied to the generation of test cases for robot software components by simulation. A test case for a robot software component consists of an input vector that requires Test Data of Input (TDI), Test Data of Simulation Dependency (TDSD), and Test Data of Test Stub (TDTS) because the behavior of CUT is affected by these test data. TDI is an input parameter of the interface under testing. Because TDSD refers the simulation control data for interface testing, it is the data affecting the CUT through the simulation. TDTS is an output data from a required interface of the CUT.

The process for the automatic interface test case generation has two steps, which is shown in Fig. 2.

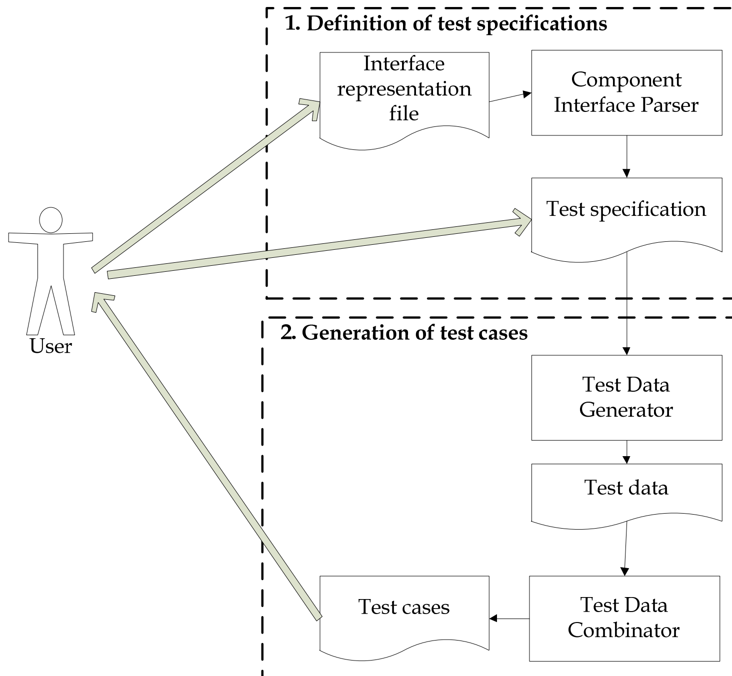


Fig. 2. Process of the automatic interface test case generation

3.1.1 Definition of test specifications

The Interface Parser module in the Interface Test Case Generator parses the interface representation of a CUT and the information on input test parameters of CUT is extracted. Fig. 3 shows a simplified XML schema of the interface representation. The main elements of the interface representation are *type_name*, *method*, and *param*. The *type_name* represents type of the interface and the *method* and *param* describe prototype of methods in the interface.

The test specification includes valid range values, specific candidate values, pre/post-conditions, and other values, for each test parameter. And a test model is a set of the test specifications, as shown in Fig. 4. The essential elements of the test model are *name*, *input_spec_list*, and *order_of_combinations*. The *name* is a name of the test model, as identifier. The *input_spec_list* includes test specifications of TDI, TDSD, and TDTS. The o*rder_of_combinations* is a number of interaction strength for pairwising the each test parameters.

In Fig. 4, the plus sign (+) indicates that the element further consists of *test_element* which is used to generate the test data values. The *test_element* is a key element in the test specification, which contains information on a test parameter such as type of the parameter, specific candidate value, and the method for test data generation. The *type* in *test_element* presents the type of a test parameter such as TDI, TDSD, and TDTS. The *test_spec* describes a method and additional information for generating test data, and has different XML schema for each type of *test_spec*, as shown in Fig. 5, Fig. 7, and Fig. 8.

```
<xs:schema>
   <xs:element name="service_port_type">
     <xs:complexType>
         <xs:sequence>
          <xs:element name="type_name" type="NCName"/>
          <xs:element ref="method" maxOccurs="unbounded" />
       </xs:sequence>
     </xs:complexType>
  </xs:element>
   <xs:element name="method">
     <xs:complexType>
         <xs:attribute name="name" type="xs:NCName" use="required"/>
         <xs:attribute name="return_type" type="xs:NCName" use="required"/>
         <xs:attribute name="call_type" type="xs:NCName" default="blocking"/>
       <xs:sequence>
          <xs:element maxOccurs="unbounded" ref="param"/>
       </xs:sequence>
     </xs:complexType>
  </xs:element>
   <xs:element name="param">
     <xs:complexType>
         <xs:sequence>
          <xs:element name="type" type="xs:NMTOKEN"/>
             <xs:element name="name" type="xs:NCName"/>
       </xs:sequence>
         <xs:attribute name="index" type="xs:integer" use="required"/>
     </xs:complexType>
  </xs:element>
</xs:schema>
```

Fig. 3. Simplified XML Schema of an interface representation

```
<xs:schema>
   <xs:element name="test_model">
    <xs:complexType>
     <xs:sequence>
      <xs:element name="name" type="xs:string" use="required" />
         <xs:element name="description" type="xs:string/>
       + <xs:element ref="pre_condition_spec_list" />
       + <xs:element ref="input_spec_list" use="required"/>
       + <xs:element ref="output_spec_list" />
       + <xs:element ref="post_condition_spec_list" />
          <xs:element name="order_of_combinations" type="xs:integer"
            use="required"/>
       </xs:sequence>
    </xs:complexType>
   </xs:element>
   <xs:element name="test_element">
    <xs:complexType>
     <xs:sequence name="name" type="xs:string" use="required"/>
       <xs:sequence name="type" type="xs:string" use="required"/>
       <xs:sequence name="description" type="xs:string"/>
      <xs:sequence ref="test_spec" use="required" />
      <xs:sequence ref="user_value_list" />
       <xs:attribute name="abstract_type" type="xs:string" use="required" />
       <xs:attribute name="real_type" type="xs:string" use="required"/>
    </xs:complexType>
   </xs:element>
</xs:schema>
```

Fig. 4. Simplified XML Schema of a test model

3.1.2 Generation of test cases

The Test Data Generator (TDG) module in the ITCG generates the test data satisfying the test specification for each test parameter. The TDG automatically generates the numeric test data by applying an equivalence partitioning scheme (ECP), a boundary value analysis scheme (BVA), and a random testing scheme. Furthermore this paper generates the test data of string type using BVA and random testing scheme.

The ECP scheme (Ostrand & Balcer, 1998) is a software testing technique that divides input data for a software unit into partitions of data from which test cases cane be derived. In principle, test cases are designed to cover each partition at least once. This technique aims to define test cases and uncover classes of errors, thereby reducing the total number of test cases that must be developed. Additionally this paper defines types of equivalence class, listed in Table 1. The TDG automatically generates test data by each type of equivalence class. Fig. 5 shows simplified XML schema of *test_spec* element for ECP.

Type of equivalence class	Description
NEC_NUMERIC_ONE_BOUDARY	The type includes just one boundary. If x < 10, there are two equivalence classes, x<10 and x>10.
NEC_NUMERIC_TWO_BOUDARY	The type includes two boundaries. If -1 < x < 10, there are three equivalence classes, x<-1, -1<x<10, x>10.
NEC_BOOLEAN	If x=true, there are two equivalence classes, x=true, x=false.
NEC_NUMERIC_CONSTANT	If x=3, there are two equivalence classes, x=3, x!=3
NEC_NUMERIC_SET	If x={-1,0,1}, there are two equivalence classes, x={-1,0,1}, x!= {-1,0,1}.

Table 1. Types of equivalence class for ECP scheme

```
<test_spec type="ECP" >
    <xs:complexType>
     <xs:element ref=" equiv_class" />
    </xs:complexType>
    <xs:element name="equiv_class">
      <xs:attribute name="type" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value=" NEC_NUMERIC_ONE_BOUDARY"/>
            <xs:enumeration value=" NEC_NUMERIC_TWO_BOUDARY"/>
            <xs:enumeration value=" NEC_BOOLEAN"/>
            <xs:enumeration value=" NEC_NUMERIC_CONSTANT"/>
            <xs:enumeration value=" NEC_NUMERIC_SET"/>
          </xs:restriction>
        </xs:simpleType>
    </xs:attribute>
    …….
    </xs:element>
</test_spec>
```

Fig. 5. Simplified XML schema of *test_spec* element for ECP scheme

The BVA scheme (Hoffman et al., 1999) is a software testing technique that designs tests including representatives of boundary values. Values on the minimum and maximum edges of an equivalence partition are tested. The values could be input or output ranges of a software component. Boundaries are common locations for errors that result in software faults, so they are frequently explored in test cases. Furthermore this paper defines the offset value of boundary for generation of elaborate test data, as shown in Fig. 6. This paper automatically generates the test data by the BVA such as the values of min_low_off_set, minimum boundary, min_high_off_set, max_low_off_set, maximum boundary, max_high_off_set, and additionally a middle value.

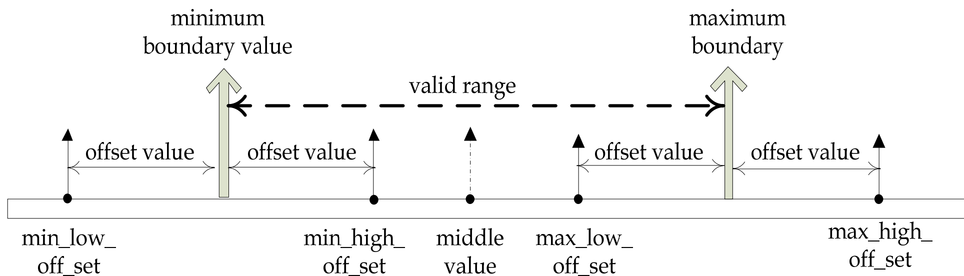


Fig. 6. Offset and test data of BVA scheme

In the BVA, there are just two types of equivalence class because the scheme needs range values, as shown in Fig. 7.

```
<test_spec type="BVA" offset="" >
    <xs:complexType>
     <xs:element ref=" equiv_class" />
    </xs:complexType>
    <xs:element name="equiv_class">
      <xs:attribute name="type" use="required">
        <xs:simpleType>
          <xs:restriction base="xs:string">
            <xs:enumeration value=" NEC_NUMERIC_ONE_BOUDARY"/>
            <xs:enumeration value=" NEC_NUMERIC_TWO_BOUDARY"/>
          </xs:restriction>
        </xs:simpleType>
   </xs:attribute>
   …….
   </xs:element>
</test_spec>
```

Fig. 7. Simplified XML schema of *test_spec* element for BVA scheme

A random testing scheme (Ntafos, 1997, Zhu, 1997) is a strategy that requires the "random" selection of test cases from the entire input domain. For random testing, values of each test case are generated randomly, but very often the overall distribution of the test cases has to conform to the distribution of the input domain, or an operational profile. In this paper, the

scheme is used to generate test data of numeric and string types. In particular, we combine the random testing scheme and the BVA scheme for generation of test data of string type. This paper analyses the boundary value of minimum and maximum length of string, and then randomly generates test data, which is satisfied with the options such as alphabet, number, special character, space, and negative character, as shown in Fig. 8.

```
<test_spec type="RANDOM_STRING">
   <xs:complexType>
    <xs:sequence>
     <xs:element name=”min_length” type=”xs:integer” use=”required” />
       <xs:element name=” max_length” type=”xs:integer” use=”required” />
       <xs:element name=” alphabet” type=”xs:boolean” use=”required” />
       <xs:element name=” number” type=”xs: boolean” use=”required” />
       <xs:element name=” special_char” type=” xs:boolean” use=”required” />
       <xs:element name=” space” type=”xs:boolean” use=”required” />
       <xs:element name=” negative_char_list” type=”xs:string” />
    </xs:sequence>
   </xs:complexType>
</test_spec>
```

Fig. 8. Simplified XML schema of *test_spec* element for random string scheme

The Test Data Combinator (TDC) module in ITCG combines the test data using a pairwise scheme (Williams, 2000) for reducing the number of test cases. The pairwise scheme is an effective test case generation technique, which is based on the observation that most faults are caused by interactions among input vectors. The TDC enables two-way combination, three-way combination, and all possible combinations of the test data, which allows the user to remove overlapping test cases from pairs of parameter combinations. The combined test cases are stored in an XML file, as shown in Fig. 9. The *case_param_info* describes name and type of a test parameter and the *case_list* consists of values of the test case.

```
<xs:schema>
   <xs:element name=”test_suite_data”>
    <xs:complexType>
     <xs:sequence>
      <xs:element name=”name” type=”xs:string” use=”required” />
        <xs:element name=”description” type=”xs:string/>
        <xs:element name=”test_ model_name” type=”xs:string” />
      + <xs:element ref=”case_param_info” />
      + <xs:element ref=”case_list” />
       </xs:sequence>
    </xs:complexType>
</xs:schema>
```

Fig. 9. Simplified XML schema of the test cases

3.2 Automatic test execution by simulation

The test application performs testing by interacting with the robot simulator, as shown in Fig. 10. This paper automatically generates skeleton source codes for test applications and links the components to each other for simplifying testing.

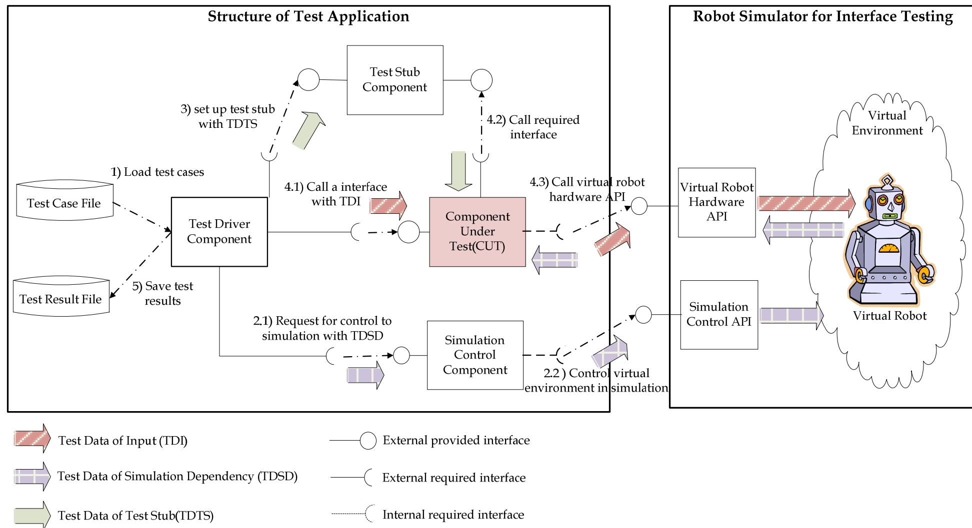


Fig. 10. Structure and operation sequence of a test application

The test application consists of a Test Driver (TD) component, a Simulation Control (SC) component, a Test Stub (TS) component, and a CUT, as shown in Fig. 10. The TD component controls the overall operation of test. During the testing, the TD component reads test cases, and sets the simulation environment and the required interface of the CUT. After the end of testing, the component stores the test results in a file. The SC component sets the simulation environment through the simulation control API in the robot simulator. TS component provides virtual interfaces of the same type as the required interface of the CUT. The TS component simulates the behavior of CUT-dependent software components. Thus, the component is used instead of an actual software component which is needed for execution of the CUT.

The test application and the robot simulator are connected to each other and the following operations shown in Fig. 10 are performed to test the CUT: 1) Read test case file, 2) Call the interface of SC component for control to the simulation environment, 3) Set up the TS component using the TDTS values, 4) Call the interface of the CUT, 5) Save the test results in a file.

The TD component reads a test case file and divides it into the TDI, the TDSD, and the TDTS. The TD component calls the interface of SC component using the TDSD values for setting the simulation environment. The SC component changes the virtual test

environment through the simulation control API using TDSD values. After the virtual test environment setup is completed, the TD component set up the output of the required interface of the CUT via the TS component interface using the TDTS. After the configuration of the simulation environment and the TS component are completed, the TD component calls interface of the CUT using the TDI as the input parameters. The CUT calls the interface of the TS component and requests or receives data via the robot hardware API during the simulation. When the operation is completed, the value resulting from the operation may be returned to the TD component. The TD component compares the actual resulting value with the expected resulting value and saves the test result in a file. After all of the testing, the TBA uploads the test result file and the test log file to ITAES. The test log file contains log information on compilation and execution of the test application.

Fig. 11 shows simplified XML schema of the test result descriptor. The *summary* consists of the number of pass and fail, and information on processing times of the interface of the CUT such as a minimum time, a maximum time, an average time, and a standard deviation time. The *test_result_list* contains detailed information on the test result, such as expected and actual test results and a processing time of the interface of the CUT, for each test case. The ITAES reads the test result file and log files, and then shows the information through web interfaces which are table-based view and graphic-based view, for easily analysing the test result of the CUT.

```
<xs:schema>
   <xs:element name="test_result_descriptor">
     <xs:complexType>
       <xs:sequence>
           <xs:element name="name" type="xs:string" />
         + <xs:element ref="summary" />
         + <xs:element ref="test_result_list" />
         </xs:sequence>
     </xs:complexType>
   </xs:element>
</xs:schema>
```

Fig. 11. Simplified XML schema of a test result descriptor

4. Evaluation

In this section, the proposed framework is evaluated using an example of the test of the Infrared Ray (IR) sensor component interface.

This paper implements the Interface Testing Automation Engine Server (ITAES) and Test Build Agent (TBA) in Java and Flex. And the robot simulator used in this paper is OPRoS simulator (http://www.opros.or.kr/). This paper develops the test simulation environment for testing shown in Fig. 12. The environment consists of an IR sensor robot which has some virtual IR sensors linked to the IR sensor simulation API and an obstacle which can move by the obstacle distance control API.

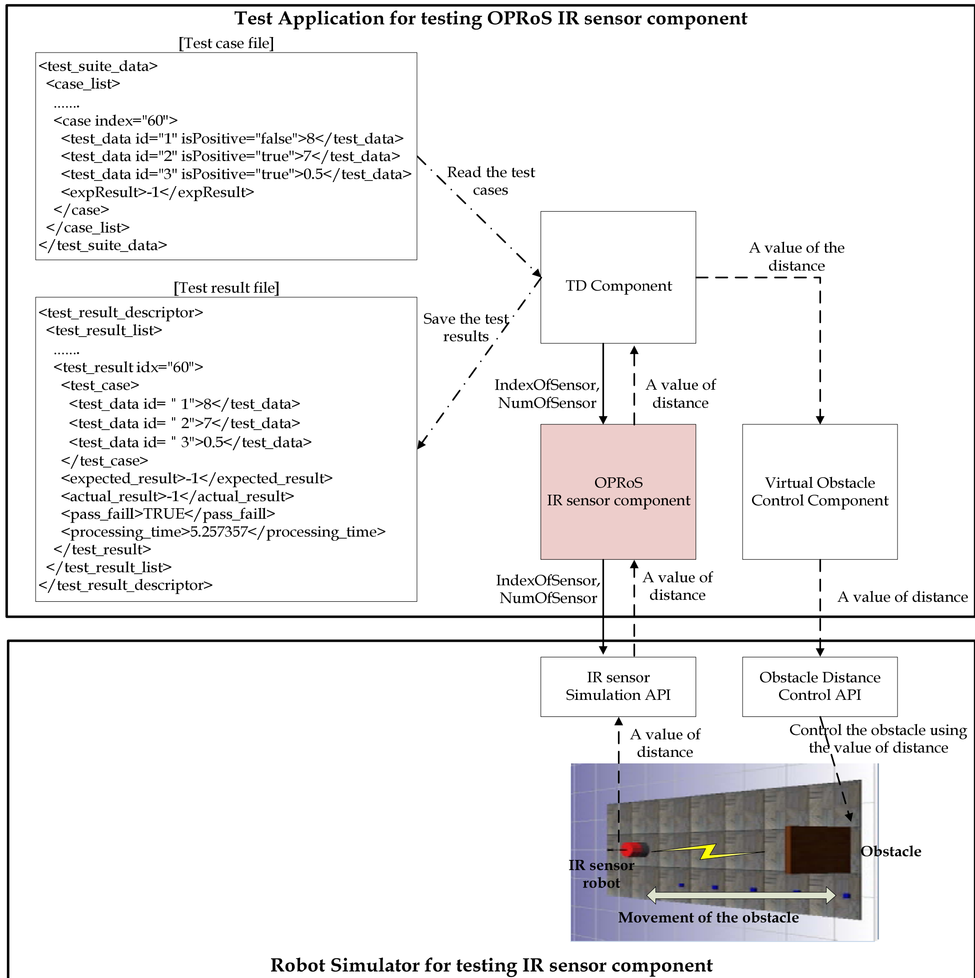


Fig. 12. Application for testing the IR component interface

We tested an Open Platform for Robotic Service (http://www.opros.or.kr/) with the Infrared Ray (IR) sensor component interface, named *GetInfraredData* interface, which had two input parameters of *IndexOfSensor* and *NumOfSensor*. The function of the interface is to get a distance value using IR sensors. This paper defines a new test parameter for TDSD, named “Distance”, to control the virtual obstacle in the test simulation environment. Thus, there are two input test parameters and one simulation-dependent parameter, which are shown in Table 2.

Name	Type	Test Specification	Description
IndexOfSensor	TDI	1 <= IndexOfSensor <= 7, The offset value is 1.	It is index of IR sensor.
NumOfSensor	TDI	1 <= NumOfSensor <= 7, The offset value is 1.	It is the number of IR sensor.
Distance	TDSD	0.0 <= #Distance <= 10.0, The offset value is 0.5.	It is a distance between an IR sensor and an obstacle in the test simulation environment. The obstacle is moved using the value of the distance test parameter.
Expected Return Value		If all test parameter are valid values, the value is same of the distance value. If the test parameters are not valid values, the value is -1.	-1 means the operation is failed.

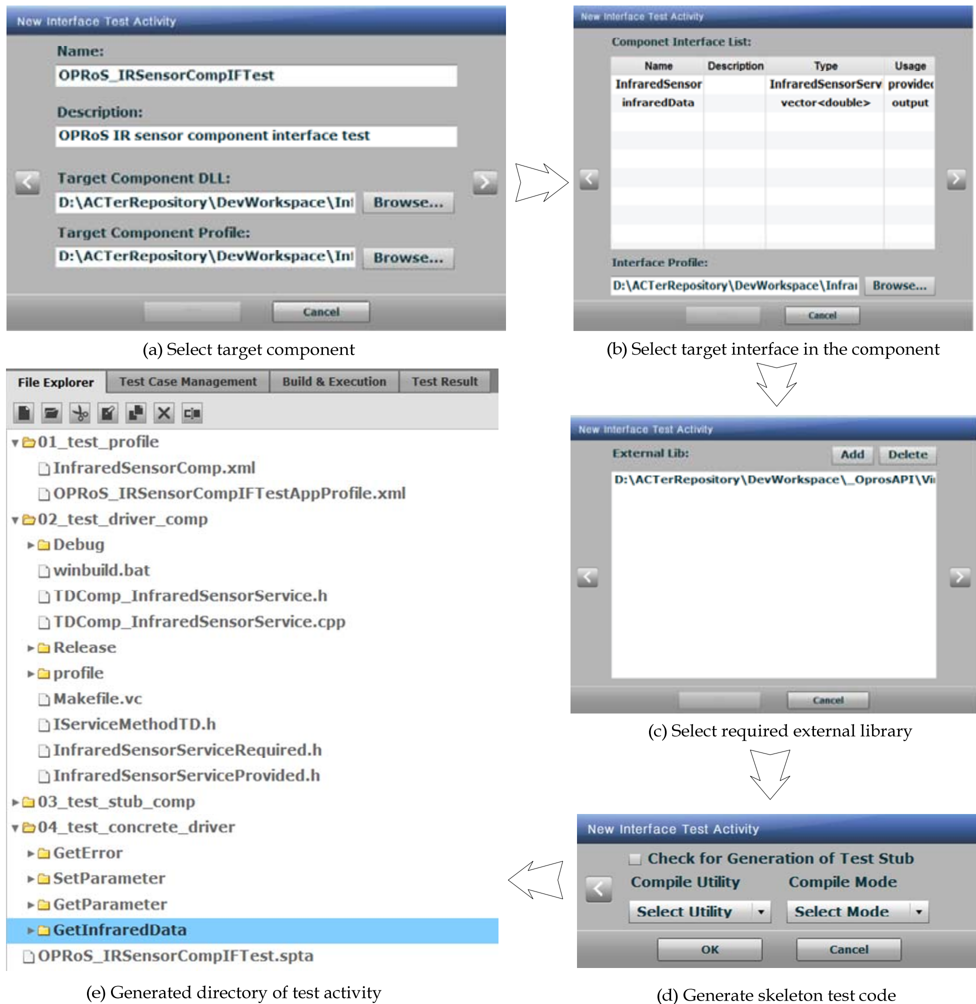
Table 2. Test Specifications and an expected test result of *GetInfraredData* Interface

The test application contains the Test Driver (TD) component, OPRoS IR sensor component, and the virtual obstacle control component. The test application does not have a test stub component because the OPRoS IR sensor component does not have a required interface. First the TD component moves the virtual obstacle in the simulation via the virtual obstacle control component using the value of *Distance* test parameter. Then the TD component calls the interface of the IR sensor component using values of the *IndexOfSensor* and the *NumOfSensor* test parameter. If the return value of the interface of the IR sensor component is same of *Distance* value, the test case is a success.

We validate three functions of proposed framework, which are the creation of test activity, the automated test case generation, and the automatic test execution by simulation.

The process of the creation of test activity has four steps, which are shown in Fig. 13(a) – Fig. 13(d). In the creation of the test activity, the information on the IR sensor component such as component profile, dll file, and interface profile, external library (optional), and the type of the skeleton test code are used.

Fig. 13(e) shows the generated directory of the test activity after completion of the process. The directory includes the test profile, the test driver component, the test stub component, and the concrete test driver.



(a) Select target component

(b) Select target interface in the component

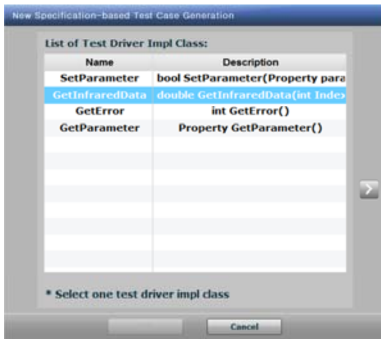
(c) Select required external library

(d) Generate skeleton test code

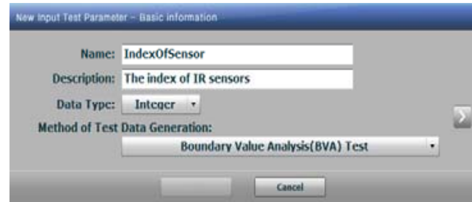
(e) Generated directory of test activity

Fig. 13. Process of creating the test activity for IR sensor component

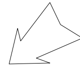
The process of automatic test case generation has four steps, which are shown in Fig. 14(a) – Fig. 14(d). We input the test specifications through the web user interface, which are the *IndexOfSensor*, the *NumOfSensor*, and the *Distance*. We analyse valid range value of each test parameter using BVA scheme. And then we select "2-way", as order of combination for pairwise. After completion of the process, 60 test cases are generated as shown in Fig. 15. Then we input the expected test result into each test case.

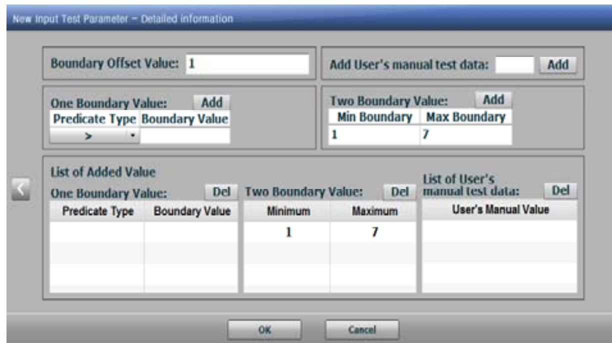


(a) Select detailed target interface

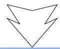


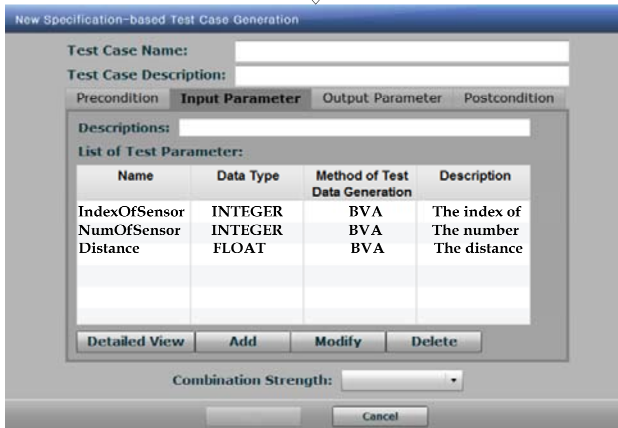
(b) Add test parameter





(c) Input detailed test specification





(d) List of the added test parameters and select the order of combinations for pairwise

Fig. 14. Process of automatic test case generation of *GetInfraredData* interface

TDI | TDSD | Expected Result

Case Information

No.	IndexOfSensor	NumOfSensor	Distance	Expected
1	7	4	9.5	-1
2	1	4	5.0	5.0
3	6	1	0.5	-1
4	4	4	0.5	0.5
5	6	4	0.0	-1
6	1	1	0.0	0.0
7	1	2	10.0	10.0
8	7	1	10.0	-1
9	6	7	5.0	5.0
10	2	6	5.0	5.0
11	4	2	5.0	-1
12	1	6	9.5	9.5
13	2	2	0.5	0.5
14	6	6	10.0	10.0
15	2	1	9.5	-1
16	2	4	10.0	10.0
17	7	2	0.0	-1

OK Cancel

Fig. 15. Test Case list of *GetInfraredData* interface

We perform and evaluate the example of test application for the IR sensor component in the test simulation environment, as shown in Fig. 12. Finally we compare the return value of the interface of the IR sensor component with the value of expected test result. Fig. 16 shows C++ code of test driver component used in this evaluation.

```
void IRConcreteTestDriver::runTest(int testCaseID){

   // Read test case
   MethodTestCase* testCase = m_methodTestCaseList->getMethodTestCase(testCaseID);

   // Type casting of each test case data
   int tcIndexOfSensor = StringToInteger(testCase->getTestDataString(0));
   int tcNumOfSensor   = StringToInteger(testCase->getTestDataString(1));
   double tcDistance   = StringToDouble(testCase->getTestDataString(2));

   // variable for measurement of interface processing time
   double elapsedTime = 0.0;

   // Moving an obstacle in the test environment
   RobotSimulator->moveWall(tcDistance);

   // Call the IR sensor component interface
   TestUtil::start_timer(1);
   double ret = pTargetServiceObj->GetInfraredData(tcIndexOfSensor, tcNumOfSensor);
   elapsedTime = TestUtil::get_timer(1);

   // Comparing test result
   string strRealValueOfTestResult = DoubleToString(ret);;
   double expectedValueOfTestResult = StringToDouble(testCase->getExpectedTestResult());
   if (ret == expectedValueOfTestResult)
      testCase->setTestResult("PASS");
   else
      testCase->setTestResult("FAIL");
   testCase->setRealTestResultValue(strRealValueOfTestResult);

   // save the processing time of the IR sensor component interface
   testCase->setMethodProcessTime(elapsedTime);
}
```

Fig. 16. Test driver code for testing *GetInfraredData* Interface

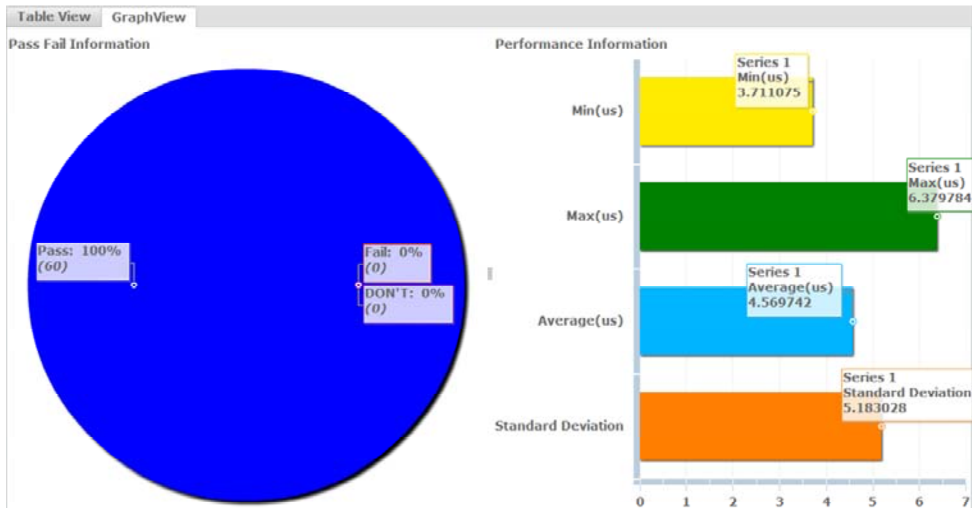
Two types of test result view are provided and shown in Fig. 17. The table-based view shows the test result of each test case. And the graphic-based view shows the summary test result, including the percentage of test success and the processing time of the IR sensor component. The example used shows that the proposed framework is working well.

Table View | GraphView

Number of Total Cases : 60 Min Time(us) : 3.711075 Statement(%) :
Number of PASS : 60 Max Time(us) : 6.379784 Branch(%) :
Number of FAIL : 0 Average(us) : 4.569742
Number of DON'T : 0 Std Deviation(us) : 5.183028

No.	IndexOfSensor	NumOfSensor	Distance	ER	MR	PF	Exetime
1	7	4	9.5	-1	-1	TRUE	4.605137
2	1	4	5.0	5.0	5.0	TRUE	6.494382
3	6	1	0.5	-1	-1	TRUE	5.875869
4	4	4	0.5	0.5	0.5	TRUE	5.566613
5	6	4	0.0	-1	-1	TRUE	5.566613
6	1	1	0.0	0.0	0.0	TRUE	4.020332
7	1	2	10.0	10.0	10.0	TRUE	5.875869
8	7	1	10.0	-1	-1	TRUE	5.257357
9	6	7	5.0	5.0	5.0	TRUE	5.257357
10	2	6	5.0	5.0	5.0	TRUE	4.948101
11	4	2	5.0	-1	-1	TRUE	4.948101
12	1	6	9.5	9.5	9.5	TRUE	6.494382
13	2	2	0.5	0.5	0.5	TRUE	5.257357
14	6	6	10.0	10.0	10.0	TRUE	5.875869
15	2	1	9.5	-1	-1	TRUE	5.257357
16	2	4	10.0	10.0	10.0	TRUE	5.875869
17	7	2	0.0	-1	-1	TRUE	5.566613
	Test cases			Expected results	Actual results	Pass(true) /Fail(false)	Processing time

(a) Table view of test result



(b) Graphic view of test result

Fig. 17. Test result views of *GetInfraredData* interface

5. Conclusions and future works

This paper proposes and develops the interface testing framework, SITAF, based on simulation and specification-based test for robot software components and develops the automatic test case generation technique for interface testing. SITAF uses three types of the test parameters, which are the input parameter, the test stub parameter, and the simulation dependent parameter and applies specification-based test techniques. SITAF also performs the automatic interface testing to identify errors caused by CUT interactions with an external environment.

SITAF is evaluated via the example of the test of the IR sensor component used in the distance measurement. The example shows that the SITAF generates test cases and performs the automatic interface testing by interactive simulation.

As future works, are considering an automatic regression test by applying software configuration management, and the mixed test environment of a simulation-based environment and a real environment for testing robot software component.

6. References

A. Martin, A. & Emami, M. R. (2006). An Architecture for Robotic Hardware-in-the-Loop Simulation, *Proceedings of the International Conference on Mechatronics and Automation*, pp.2162-2167, June 2006

Bundell, G. A.; Lee, G., Morris, J. & Parker, K. (2000). A Software Component Verification Tool, *Proceedings of the Conf. Software Methods and Tools*, pp. 137-146, 2000

Buy, U.; Ghezzi , C., Orso , A., Pezze, M. & Valsasna, M. (1999). A Framework for Testing Object-Oriented Components, *Proceedings of the 1st International Workshop on Testing Distributed Component-Based Systems*, 1999

Edwards, S. H. (2001). A framework for practical, automated black-box testing of component-based software, *International Journal of Software Testing, Verification and Reliability*, Vol. 11, No. 2, pp. 97-111, June, 2001

Hoffman, D.; Strooper, P. & White, L. (1999). Boundary Values and Automated Component Testing, *Journal of Software Testing, Verification, and Reliability*, Vol. 9, No. 1, 1999, pp. 3–26,

http://www.opros.or.kr/

Hu, X. (2005). Applying Robot-in-the-Loop-Simulation to Mobile Robot Systems, *Proceedings of the 12th International Conference on Advanced Robotics*, pp. 506-513, July 2005

Jawawi, D.N.A.; Mamat, R. & Deris, S. (2007). A Component-Oriented Programming for Embedded Mobile Robot Software, *International Journal of Advanced Robotics Systems*, Vol.4, No.2, 2007, pp.371-380, ISSN 1729-8806

Michel, O. (2004). Webots: Professional Mobile Robot Simulation, *International Journal of Advanced Robotic Systems*, Vol. 1, No. 1, 2004, pp. 39-42, ISSN 1729-8806

Momotko, M. & Zalewska, L. (2004). Component+ Built-in Testing: A Technology for Testing Software Components, *Foundations of Computing and Decision Sciences*, pp. 133-148, 2004

Ntafos, S. (1998). On Random and Partition Testing, *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, 1998, pp. 42–48

Ostrand, T.J. & Balcer, M.J. (1998). The Category-Partition Method for Specifying and Generating Functional Tests, *Communications of the ACM*, Vol. 31, No. 6, 1988, pp. 676–686

Williams, A.W. (2000). Determination of test configurations for pair-wise interaction coverage, *Proceedings of the 13th Conf. Testing of Communicating Systems*, pp. 59-74, August 2000

Zamli, K. Z.; Isa, N. A. M., Klaib, M. F. J. & Azizan, S. N. (2007). A Tool for Automated Test Data Generation(and Execution) Based on Combinatorial Approach, *International Journal of Software Engineering and Its Applications*, Vol. 1, No. 1, pp. 19-35, July, 2007

Zhu, H.; Hall, P.& May, J. (1997). Software Unit Testing Coverage and Adequacy, *ACM Computing Surveys*, Vol. 29, No. 4, December 1997, pp. 366–427

© 2012 The Author(s). Licensee IntechOpen. This is an open access article distributed under the terms of the Creative Commons Attribution 3.0 License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited.