Aspect-based On-the-Fly Testing Technique for Embedded Software

Jong-Phil Kim, Jin-Soo Park and Jang-Eui Hong
Department of Computer Science, Chungbuk National University, Cheongju, Rep. of Korea

Keywords: Aspect-oriented Programming, On-the-Fly Testing, Embedded Software.

Abstract: Various techniques for testing embedded software have been proposed as a result of the increased need for high quality embedded systems. However, it is hard to perform accurate testing with these techniques on failures that can occur unexpectedly in a real environment, because most of the tests are performed in software development environment. Therefore, it needs a testing technique that can dynamically test software's latent faults in a real environment. In this paper, we propose an aspect-based On-the-Fly testing. The purpose of which is to test the functionalities and non-functionalities of embedded software using aspect-oriented programming at run-time in a real environment. Our proposed technique provides some advantages of prevention of software malfunction in a real environment and high reusability of test code.

SCIENCE AND TECHNOLOGY PUBLICATIONS

1 INTRODUCTION

Many types of embedded software, especially aerospace software and national defense software are tested on functionality and non-functionality in a development environment as well as in an operational environment. These tests are expensive and require much time and heavy test harness (Michael, 2006). In spite of such testing, embedded software malfunctions frequently occur in real operation. To address these problems, a built-in testing approach and aspect-based testing approach have been attempted. However, the built-in testing was performed for interface conformance testing to integrate a reusable component in component-based development, and aspect-based testing was performed to test the functionality of software during development. Therefore, the existing research has various limitations to test unexpected behaviors that can happen to the run-time of embedded software.

In this paper, we propose an On-the-Fly testing technique to perform self-testing while embedded software is executed in a real environment. This technique can perform testing in a real operation environment including some test cases that are not covered in development testing. Our proposed testing technique can provide a method to prevent malfunction that can happen during real operation of a system, since the technique uses aspect components defined with aspect-oriented programming concepts to test embedded software. Using the aspect components, we can develop separately the function code of embedded software and the test code. The test code is installed in the target system by weaving with the function code of the embedded software. The test code can inspect that execution of a program code is performed correctly at run-time and can prevent the occurrence of unexpected failures, which were not detected in host-based testing. In this way, our testing technique provides advantages of prevention of software malfunction in a real environment and high reusability of test code.

The paper is organized as follows. Section 2 surveys related work about existing aspect-based testing techniques. Section 3 explains the issues that we want to test in our research. In section 4 we design aspect components to test the issues described in section 3. We present a process to perform On-the-Fly testing using aspect components, and show a case study of our approach in section 5. The paper concludes with a summary of our research and directions for future work in section 6.

2 RELATED WORK

To test whether reusable components have the correct functionalities, an aspect with Built-in testing characteristics is proposed (Jean et al, 2003). In

Kim J., Park J. and Hong J..
Aspect-based On-the-Fly Testing Technique for Embedded Software.
DOI: 10.5220/0004081503750380
In *Proceedings of the 7th International Conference on Software Paradigm Trends* (ICSOFT-2012), pages 375-380
ISBN: 978-989-8565-19-8
Copyright © 2012 SCITEPRESS (Science and Technology Publications, Lda.)

(Dehla and Matthias, 2003), they developed an aspect to test the problem of inheritance or information concealment etc. in object-oriented programming. In (Martin and Cristina, 2000), they defined an aspect to inspect grammatical difference (i.e. difference of data type) between unit modules. Also, in (Fernando et al, 2007), they defined an aspect to perform error handling. In (Jani, 2006), he tested mobile operating systems by weaving a testing aspect into the Symbian operating system. However, these researches focused on development testing and the role of a test agent or a test oracle.

Our proposed testing technique is able to perform on-the-fly testing in a real operation environment using aspect components which can independently perform a test by itself and can control the behavior of software with the testing results.

3 ISSUES OF ON-THE-FLY TESTING

Software testing is generally categorized into functional testing and non-functional testing. In particular, embedded software is important to test non-functional issues such as platform portability, memory constraints etc. as well as normal functionality (Mirko et al, 2005). Figure 1 shows issues to be tested for embedded software.

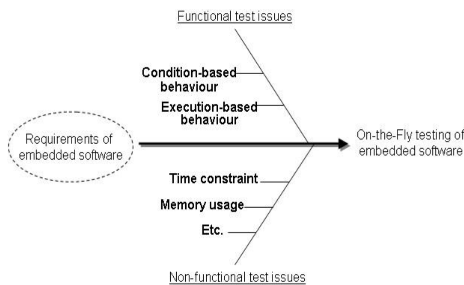

Figure 1: Testing issues of embedded software.

3.1 Functional Testing Issues

3.1.1 Condition-based Behavioural Test

The condition-based behavioural test is to test the pre-condition or post-condition of a specific module during execution. This inspects whether input events or state variables of a target module satisfy specific conditions. Figure 2 shows the procedure of a condition-based behavioural test. In order to explain the procedure, we define the following basic functions.

- int Fun(Pi,i=1…n): function to test a target module that has n input parameters and the return value of integer type
- Get(x): function to read input variable or state variable x
- Cond(x)::={true|false}: function to inspect whether the current value of variable x satisfies a functional requirement
- Alarm(): function to notify exception occurrence
- IHR(): function to perform exception handling

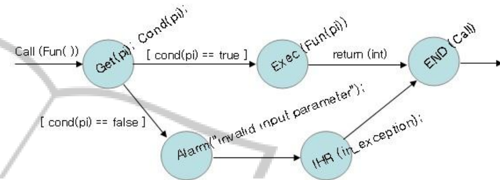

Figure 2: Procedure of pre-condition based behavioural Test.

3.1.2 Execution-based Behavioural Test

The execution-based behavioural test is to inspect the execution result of a module by analysing the log data after logging the state changes for module execution. For example, after the booting-up module of an embedded system is executed, this test can be used to confirm that the initialization of the system was executed correctly.

3.2 Non-functional Testing Issues

3.2.1 Time Constraint Test

The time constraint test is to inspect whether the execution time of a specific module or a method satisfies the time constraint specified in the requirements. Figure 3 shows the procedure of the time constraint test. To describe the test procedure, we define the following a basic function. Other functions have the same behaviour as the condition-based behavioural test.

- Set(T): function to save current time clock

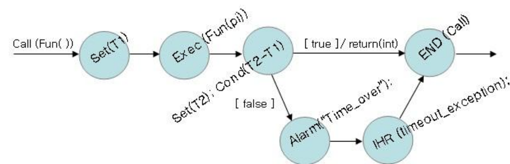

Figure 3: Procedure of time constraint test.

3.2.2 Memory Usage Test

The memory usage test checks that the available

memory of the system is enough to execute a module. If there is more available memory than required memory, this executes the relevant function. Otherwise it exits executing the function. Such behaviours are represented in Figure 4.

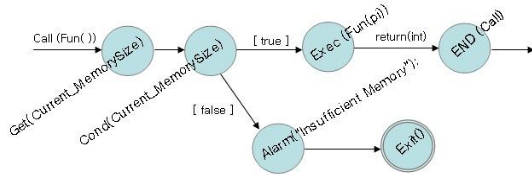

Figure 4: Procedure of memory usage test.

4 DESIGN OF ASPECT COMPONENT

4.1 New Pointcut for Non-functional Testing

The existing aspect-oriented programming (AOP) languages provide pointcut designators that pick out a point in execution flow, for example, when a method is called. Advices are executed at that join points which is defined by pointcut designators. However, a non-functional testing such as time constraint test and memory usage test can need advices that are called when a specific code block is executed because it is necessary to test a non-functional requirement in a specific behavior area. The existing AOP languages do not allow developers to pick out the area in time while a code block is executed.

To address this limitation of AOP, we define a new pointcut, which we call "testing pointcut". As shown in Figure. 5, the testing pointcut picks out a join point which is the code block selected by a tester.

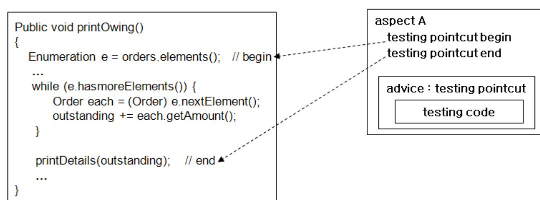

Figure 5: Testing pointCut for non-functional testing.

For example, in Figure 5, the testing pointcut of aspect A specifies the call to orders.elements() as the beginning of the testing pointcut and the call to printDetails() as the end of it. So, it is able to perform a selective non-functional testing for time-critical or memory usage-critical code block, because all statements within a code block specified by the testing pointcut are defined as the join point. The matching of the testing pointcut is performed through control flow graph that specifies the event sequence of a target system. The following syntax defines a testing pointcut tp:

tp::=testing pointcut[call(a); call(a)]

a::=class name.method()

For example, the following testing pointcut:

```
Pointcut tp(): testing pointcut[call(*.orders.elements()); call(*.printDetails())];
```

4.2 Example System for Testing

For easy understanding of our further explanation, we introduce an example robot application with the following requirements.

> A military reconnaissance robot receives external sound from sound sensors. This robot recognizes the input sound and identifies the direction from which the sound originated. Then, the robot takes a picture of the object using a camera after a change of direction to the origin of the sound. The robot must be able to react quickly to movement of the sound (with a maximum of one second until turning of direction).

4.3 Functional Behaviour Test Component

The functional behaviour test checks whether function codes can be executed correctly. This test can examine the pre-condition and post-condition of the test target using "before" or "after" advice.

For example, a malfunction of the example robot system can happen when input data transmitted by a sound sensor of the robot is out of a predefined data range. Therefore, we define an aspect to examine this malfunction. The aspect component is represented in Figure 6 using AspectJ (Gregor, 2001). This aspect tests whether the value of decibel inputted from a sensor exists between 80dB and 120dB. If the decibel value is out of the allowed range, the aspect logs the status and exits the execution of the target module.

4.4 Time Constraint Test Component

General embedded software has time constraints in its functional execution. Because satisfying the time constraints is an important issue of embedded software, it is critical to test the time constraints in a

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

```
aspect SensorInputTest {
    pointcut InputRange(int decidel) : Call(void
                    Sensor.rvc(int decibel, int hertz);

    before (int decibel) : InputRange (decibel) {
        if ( decibel < 80 ||  decibel > 120 )  {
             Logger.entry ("input range is invalid");
             System.exit(0);
        } else
             Logger.entry ("input range is valid");
    }
}
```

Figure 6: Aspect component to test functional behaviour.

```
aspect TimeConstraintTest {
    pointcut timeoutCheck(int decibel) : call soundDirection(int decibel);
    private Timer timer = new Timer();
    long sTime, eTime, tTime;

    before(int decibel) : timeoutCheck(int decibel) {
         sTime = timer.start();
    }

    after(int decibel) : timeoutCheck(int decibel) {
        eTime = timer.stop();
        String str = Integer.toString(eTime - sTime);
        if ( (eTime - sTime) > 1000 ) {
            throw new illegalException("Time is over");
            LCD.drawString("Actual Time:"+str, 0, 0);
            LCD.drawString("Result: invalid", 0, 3);
            System.exit(0);
        } else {
            LCD.drawString("Actual Time:"+str, 0, 0);
            LCD.drawString("Result : valid", 0, 3);
          }
    }
}
```

Figure 7: Aspect component to test time constraint.

real environment. Figure 7 shows an aspect component to test a requirement of time constraints. This aspect tests the time requirement that the turning action to change the direction of the robot toward sound direction should be performed within 1000ms in our robot application. That is, if the change of direction is not executed within 1000ms, the test result is invalid. The “before” advice of this aspect sets up the timer before the soundDirection() method is called and the “after” advice is used to test time over after the method is executed.

4.5 Memory Usage Test Component

The embedded software that should be operated with a restricted resource can lead to invalid action when the system does not have enough memory for execution. Thus, if the system cannot provide the memory that is needed for execution, the system should exit the execution or wait until enough memory to execute is guaranteed. Memory usage required in the execution of a specific module can be analysed using a tool such as Eclipse Profiler and the analysed result is used to test the condition of memory usage in the advice of aspect component. Figure 8 shows an aspect component to test memory usage using the testing pointcut, which specifies selected code block for test target. The role of this aspect component is to stop the operation of the system when the current memory size of the system is smaller than 19,900 bytes.

```
aspect MemoryUsageTest
{
   pointcut availableMemory() : testing pointcut[call(Sensor.soundDirection(int decibel));
                                 call(Controller.setDirection(int direction)]

   private Runtime runtime = Runtime.getRuntime();
   private String strMemory = null;

   around () : availableMemory(I) {
     strMemory = Integer.toString(runtime.freeMemory());

     if (runtime.freeMemory() < 19900) {
          Logger.entry("Memory size is Low");
          LCD.drawString("Actual Memory Size:"+strMemory, 0, 0);
          LCD.drawString("Result: invalid", 0, 3);
          System.exit(0);
      }
      else {
           LCD.drawString("Actual Memory Size :"+strMemory, 0, 0);
           LCD.drawString("result : valid", 0, 3);
           proceed();
      }
   }
}
```

Figure 8: Aspect component to test memory usage.

5 ASPECT-BASED DYNAMIC TESTING

To test embedded software using aspect components, a source code and testing aspect should be woven with each other through compiling. The woven execution file can examine and cope with exceptional cases that can occur while the system is running. We define “On-the-Fly testing” with such dynamic testing while the system is running.

5.1 On-the-Fly Testing Process

If the aspect component arrivals statements that is described at join point, while the function modules are executed sequentially in a real environment, the aspect component starts the testing of the function module. This aspect tests whether the corresponding module may perform the correct action or whether the input parameters for the function module are valid. According to the test result, it is determined whether the function module should be executed. Therefore, our On-the-Fly testing can dynamically cope with those cases that cannot be detected in development testing. Figure 9 shows the On-the-Fly testing process of embedded software.

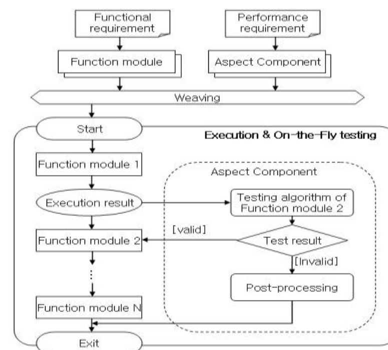

Figure 9: On-the-fly testing process.

5.2 Applying Aspect to Example System

To test the requirements (i.e. testing issues) of our example system, we apply three aspects, designed in section 4. The experiment for each test issue is performed by weaving only a single aspect for the adequate examination of a single requirement.

5.2.1 Experiment Environment

The test environment for aspect-based On-the-Fly testing is developed using JDK 1.6.0, Eclipse 3.6, AspectJ 2.1.0 and Aspect Bench Compiler (Avgustinov, 2005). We extended the AspectJ with the Aspect Bench Compiler for the new pointcut. Also, an assistance tool is used, Eclipse Profiler 0.5.33 (Eclipse, 2007), to measure memory usage. The target machine for our robot application is Mind Storm Robot NXT #9797 that has 32bits ARM7 processor and 64K RAM. Figure 10 shows the experiment environment to test our example system. In order to do an experiment, we developed function code using NXJ Java language and aspect components using AspectJ. These are executed after installing in the brick board of NXT and the experiment result is displayed on the LED of the Robot.

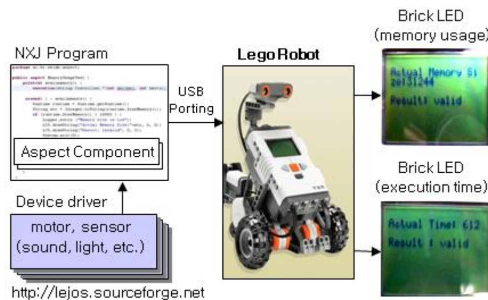

Figure 10: Experiment environment of example system.

5.2.2 Test Result of Functional Behaviour

The goal of the functional behaviour test in our experiment is to confirm that the Robot's sound sensor correctly recognizes and processes sound of fixed range. The test result of the method Sensor.rvc() using the aspect component of Figure 6 is shown in Table 1. From Table 1, two test cases t3 and t5 do not satisfied the input condition of test module Sensor.rvc().

Table 1: Test result of functional behaviour.

Test Module	Test case	Input conditions		Expected results	Actual results
		decibel	hertz		
Sensor.rvc	t1	98	1200	valid-turning	turning
	t2	85	1200	valid-turning	turning
	t3	67	1200	invalid-no change	no change
	t4	104	1200	valid-turning	turning
	t5	71	1200	invalid-no change	no change

5.2.3 Test Result of Memory Usage Test

The estimated values of memory usage are used as a test oracle for aspect component. This aspect component judges whether the amount of available memory of the system is enough or not. If the memory amount is not enough, the execution of the function module is stopped. Table 2 shows the test result of memory usage of the selected code block in our example application.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

Table 2: Test result of memory usage.

Test Module	Test case	Estimated memory	Available memory	Expected results	Actual results
Selected behavior block	t1	19900	32440	valid	valid
	t2	19900	31244	valid	valid
	t3	19900	28046	valid	valid
	t4	19900	14586	valid	invalid
	t5	19900	34228	valid	valid

The memory usage of the code block estimated by Eclipse Profiler was 19,900 bytes. The amount of available memory measured by the runtime.freeMemory() method of Java Virtual Machine is represented in the available Memory field of Table 2. When we consider the system memory size of the NXT board, all expected results are valid. However, unfortunately the actual result of test case t4 is invalid. From this result, we knew that a memory leakage occurs by memory garbage during execution of t4. Therefore execution of the selected code block is stopped at test case t4.

5.2.4 Test Result of Time Constraint

The time constraint test is to calculate the elapsed time from recognizing an input sound data to turning the direction of the robot wheel. However, the accurate elapsed time of the module execution is hard to measure because the execution can be interfered by external or internal factors. Therefore, we added statements that can measure the elapsed time within the aspect component. Table 3 shows the result of time constraint testing.

Table 3: Test result of time constraint.

Test Module	Test case	Input condition (decibel)	Estimated Time (ms)	Actual time	Expected results	Actual results
Sound Direction	t1	88	1000	610	valid	valid
	t2	97	1000	600	valid	valid
	t3	115	1000	600	valid	valid
	t4	43	1000	590	valid	valid
	t5	139	1000	610	valid	Valid

6 CONCLUSIONS

In spite of various techniques on testing of embedded software, the frequent failures of embedded systems occur in real operation environments. The cause of these failures exists fundamentally in the software itself, but the problem is that the defects for such failure are not discovered in development testing. Therefore we proposed an On-the-Fly testing technique to perform self-testing while the system is running in a real environment. The proposed testing technique can find defects missed during development testing and can prevent unexpected malfunctions that can happen during real operation of a system. In a hard real-time system, the test execution of aspect components can cause delay of response time. However, because our aspect components are very small in size and simple in complexity compared with function modules, there is no large delay on execution time. We believe that our proposed On-the-Fly testing technique gives some benefits of a real test of unexpected input conditions, prevention of software malfunction, and high reusability of test components.

Our future work is to develop techniques that can test issues such as dead lock, pre-emption scheduling and race condition between modules through extension of the aspect concept. Also, we are also going to apply our testing technique to a large-scale embedded application.

ACKNOWLEDGEMENTS

This research was supported by Next-Generation Information Computing Development Program (No.2011-0020523) and also partially supported by Basic Science Research Program (2011-0010396) through the National Research Foundation of Korea(NRF) funded by the Ministry of Education, Science and Technology.

REFERENCES

Michael, J.K., William, I.B., Carl, B.E., 2006. Effective Test Driven Development for Embedded Software. In *International Conference on Electro/Information Technology*, pp. 382-387.

Jean, M.B., Joao, A., Ana, M., Albert, R., 2003. Using Aspects to Develop Built-in Tests for Components. In *International Workshop on UML*, pp. 1-8.

Dehla, S., Matthias, V., 2003. An Aspect-Oriented Framework for Unit Testing. In *International Conference on QOSA/SOQUA*, pp. 257-270.

Martin, L., Cristina, V.L., 2000. A Study on Exception Detection and Handling Using Aspect-Oriented Programming. In *International Conference on Software Engineering*, pp. 418-427.

Fernando, C.F., Alessandro, G., Cecilia, M.F., 2007. Error Handling as an Aspect. In *International Workshop on BPAOSD*.

Jani, P., 2006. Extending Software Integration Testing Using Aspects in SymbianOS. In *International Conference on Practice Research Techniques*, pp. 147-151.

Mirko, L., Tiziana, M., Graziano, P., Bernhard, S., 2005. Dynamic and formal verification of embedded systems. *Journal of Parallel Programming*, Vol. 33, pp. 585-611.

Gregor, K., Erik, H., Jim, H., Mik, K., Jeffrey, P., William, G.G., 2001. An Overview of AspectJ. *Springer LNCS*, Vol. 2072, pp. 327-353.

Avgustinov, P., 2005. abc – an extensible AspectJ compiler. In *International Conference on AOSD*, pp. 87-98.

Eclipse, 2007. Eclipse Profiler. http://eclipsecolorer.sourceforge.net/index_profiler.html.