A CEP- and UTP-Based Testing Scheme for Event-Based Systems

Haochuan Kuang, and Taewoong Jeon

Abstract—Testing event-driven systems is not easy because there are a large number of possible event sequences that need to be monitored and analyzed for checking the correct system behavior. Test cases for an event-driven system are sequences of events rather than simple inputs. These factors make it difficult to test event-driven systems at integration level for monitoring defects in the process which we are interested in.

In this research, we describe a scheme for designing a CEP- and UTP-based test system built on JUnit framework for testing event-based systems at unit- and integration levels. The proposed testing scheme makes the testing of the event-driven system more controllable and observable.

We used UML to describe the scheme in detail. The proposed scheme is implemented using JUnit, which is a programmer-oriented testing framework for Java. We use a TankWar event-driven system as a case study for exemplifying our method.

Keywords—Complex Event Processing (CEP), UML Testing Profile (UTP), Event-driven System, JUnit, UML

I. INTRODUCTION

SOFTWARE testing accounts for a large proportion of the cost of developing software systems. Typical percentages range between 30% and 60% [1]. One of the characteristics of recent Object-Oriented (OO) software systems is the complex dependencies that exist between classes. Even a small- or reasonable-sized OO system can have very complex class relationships as measured, for instance, in terms of the number of circuits in the class dependency graph. The dependency graph may indicate that there are many ways for class instances to interact with each other [2] [3]. This situation is a challenge for any software testing at unit level and integration level.

Unit testing refers to verifying the correct behavior of a single unit within a program. The unit under test must be independent as much as possible from the other units within the system in order to prevent interference in the test results [4].

Integration testing refers to verifying the correct interactions between the units that have individually passed unit testing and integrated together [5]. Any incremental approach to software testing requires two decisions to be made: one is in what order to unit-test, integrate, and integration-test modules. The other is which testing techniques to apply to unit-, and integration testing, respectively [1]. Although there are a large number of related works on software testing, "appropriate strategies for effective incremental integration testing are required to handle the complex spectrum of possible static and dynamic dependencies between classes" [6].

Many computer systems in operations today are event-driven systems and they are getting bigger and more complex. Testing event-driven systems is not easy because there are a large number of possible event sequences that can be invoked through interfaces. Furthermore the test cases for event-driven systems are sequences of events instead of simple inputs. In the course of processing events, the systems may undergo a series of state changes [7]. All of these factors make it more difficult to test event-driven systems at the integration level for monitoring and detecting defects in the process which we are interested in.

The goal of this research is to overcome the problems stated above. We propose a scheme for designing a CEP- and UTP-based test system built on JUnit framework for testing event-based systems at unit- and integration levels without incurring changes or intervention to the source code of the target systems. The test components provided in the test system make the testing of the event-driven system more controllable and observable, and thereby enable us to effectively monitor malfunctions in the event-driven processes generated during the execution of the event-driven system.

In Chapter 2, we review some related work about CEP, UTP, the JUnit framework, and the testability of software. In Chapter 3, we describe our proposed design scheme for the CEP- and UTP-based test framework which is an extension of the JUnit. Chapter 4 presents a TankWar case study as an example in which we applied our scheme to the testing of an event-driven system. Finally, this research is concluded in Chapter 5 together with some future works.

II. RELATED WORKS

A. Complex Event Processing (CEP)

Complex event processing (CEP) defines an approach to processing events to detect further events or patterns that suggest more complicated situations. CEP offers a set of techniques and tools for understanding and controlling event-driven information systems [8]. An event in CEP is any object signifying an activity happening inside an observed system. Sometimes events within a stream of events can be
independent but they can also follow behavioral patterns that are specified by given rules. Moreover, a complex event is a higher level event which is defined by aggregation of other simple or complex events. Once a CEP system is set up to identify the patterns, rules can be triggered so that actions prescribed by the rules are taken automatically towards a pre-defined result. CEP provides a solution for monitoring and detecting whether the system behaves as expected or not at different levels of abstraction [9] [10].

**B. UML Testing Profile (UTP)**

The UML Testing Profile (UTP) extends and specializes UML to provide concepts for defining test specifications and models. UTP introduces four logical concept groups covering the aspects of test architecture, test behavior, test data and time that the original UML do not directly support. Together, these concepts define a language for modeling the artifacts of a test system [11] [12].

**C. JUnit Framework**

JUnit is a testing framework to write repeatable tests for the Java programming language. It helps in the automation of unit tests for Java classes. It is most commonly used to test the functionality of individual methods or classes at unit testing level. JUnit has played an important role in the development of Test-Driven Development. [13]

**D. Software Testability**

There are many definitions of testability. The most common is the ease of performing testing [14]. In other words, testability can be thought of as the characteristics or properties of a piece of software that make it easier to test.

Software testability can be affected directly or indirectly by many factors such as:

- **Controllability**: the ability to set up and control test conditions
- **Observability**: the ability to observe test results externally
- **Sensitivity**: the ability to capture and expose traces of malfunctions in response to tests
- **Oracle availability**: the ability to determine or obtain expected test results [15]

Testability is important for software developer and software testers because it helps them to keep the test effort under control [16].

**III. THE DESIGN SCHEME**

The design scheme embeds tester components into the test system built on JUnit in order to facilitate testing whether the SUT’s functional contracts are observed when the SUT is test-run under the test environment built by extending the test framework to adapt to testing the SUT. The design scheme is illustrated using the UML diagramming notations.

We create some types of test support components and configure them together with SUT (Software Under Test) into a test system according to the concepts of UTP to increase SUT’s testability. They are test interfaces, test emulators, test controllers, test suites and test loggers. The test interfaces specify unified abstract methods which are implemented by test emulators to avoid modifying the original codes in SUTs and to increase the controllability of SUT. The test emulators implement the test interfaces to provide expected pre-conditions, post-conditions and methods of interaction between SUTs for use by test suites. The test emulators also increase SUT’s sensitivity by capturing running traces of malfunctions in real time during the test execution. The test suites increase observability by monitoring actual results (including the ones captured by test emulators) of test execution and determine the test result’s correctness by comparing the actual with expected results (obtained from test emulators). The test controllers upgrade controllability by setting up and initializing test conditions (variables) in preparing the test environment. The final test result (passed/failed) for each test case is sent to the test loggers. The test loggers record the test result according to the current test context. The test result recorded by the loggers can be displayed in the test suites.

Fig. 1 shows the class structure of the design scheme of test framework using UML class diagrams. The test components included in the test framework are Emulators, SuiteTest, TestController, and TestLogger. The SUTs stand for the classes or components under test. The Emulators are implementation classes of the TestInterfaces, all of which provide the same interface as the SUTs. Testing at the integration testing level is started with the invocation of the methods provided by TestInterfaces. When these methods are called, each of the tester components in test framework performs its own testing function. The TestController initializes tester components and drives the test execution. In most cases, it generates instances of the SUTs and Emulators before starting the test execution. It also can initialize the TestLogger to the condition required by the test case. The Emulators calculate the expected result after the SUTs execute the methods. The correctness of the actual result will be judged by comparing the actual result obtained from the Emulators with the expected result. The TestLogger
collects and records the test results during the test execution. Following the testing pattern described above, the functional behavior between SUTs is not directly performed by themselves during the whole test process. This design pattern of the test components is similar to the Proxy design pattern. The outermost SuiteTest can be viewed as an output monitoring window, playing the role of watching the SUTs accessing the Emulators. Table I shows a concept hierarchy in CEP for our test scheme.

<table>
<thead>
<tr>
<th>Level</th>
<th>Activity Class</th>
<th>Event Types</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>SuiteTest</td>
<td>processOne, processTwo</td>
</tr>
<tr>
<td></td>
<td>EmulatorOne</td>
<td>methodOne, methodTwo</td>
</tr>
<tr>
<td></td>
<td>EmulatorTwo</td>
<td>methodOne, methodTwo</td>
</tr>
<tr>
<td></td>
<td>SUTOne</td>
<td>methodOne, methodTwo, methodThree</td>
</tr>
<tr>
<td></td>
<td>SUTTwo</td>
<td>methodOne, methodTwo, methodThree</td>
</tr>
</tbody>
</table>

There are three hierarchy levels in the SUT-integrated test system, of which level 1 is the lowest one. An activity is signified by an event of the corresponding event type. An event type is defined by a method provided by an activity class. An object of an activity class can be thought of as an event processing agent (EPA) that processes events received through method invocation. Any activity at a higher level can be specified as an aggregation of activities at the level below it, and can be processed by a group of interconnected EPAs called an EPN (Event Processing Network).

The SUTs correspond to the EPAs at level 1 and describe the internal activities of the target system. The Emulators correspond to the EPAs at level 2 and each level 2 activity is an activity of a component in the test framework. For example, Emulators can make request/response to each other through methods as illustrated in Table I. The level 3 activities consist of posets (partially ordered sets) of level 2 activities. They represent a completed transaction between components.

These event patterns can be represented as UML sequence diagrams. Fig. 2 and Fig. 3 show the UML sequence diagrams at level-3 and level 2, respectively.

In Fig. 2, the interactions between objects constitute the sequence of events at level-3 represented as interaction references to processOne and processTwo. The processOne and processTwo each aggregate the interactions among level-2 events leading to the level-3 events.

In Fig. 3, the diagram shows the sequence of events of processOne between SuiteTest, EmulatorOne and SUTOne to detect or create the level-3 event. We can further depict the interactions among level 1 events leading to the level 2 events in the similar way.

The proposed scheme in this research supports the unit- and integration testing for event-based systems. In TankWar case study, we will use the testing with the test framework for the case in which objects are associated with each other.

IV. TANKWAR CASE STUDY

A. TankWar Event-based System and Analysis

In this case study, we exemplify the proposed test framework by using a TankWar event-based system as a target for testing that we implemented in the Java language.

Fig. 4 shows the Java source files of the TankWar system. The TankServer class manages the whole control of the game in server. The ServerThread class is a thread that monitors the socket for creating new threads to process the messages received. The ServerAgentThread class is a thread that processes the messages received from a client and broadcasts these received messages to all other clients during the game playing. The TankClient class manages the whole control of the game in client. The WarMap class can construct the obstacles in the background. The Tank class is the basic tank component class that can be instantiated as tank instances during the
TankWar game playing. The Bullet class is the bullet component class that can be instantiated as bullet instances. The ClientAgentThread class is a thread that processes the messages received by a client. The ClientWar class is the main canvas that can initialize and draw the canvas. It also can perform basic functions such as clientMove() to drive a tank instance to move horizontally or vertically, tankFire(Tank tank) to drive a tank instance to fire bullets, and sendMsg(String msg) to send the corresponding status messages to the server during the game playing.

In order to observe and monitor the events which we are interested in for detecting abnormal points, we use the CEP for analyzing the target system. In our design scheme, tankServer and tankClient are SUTs at the unit testing level, and correspond to the EPAs at level 1. TankServerEmulator and tankClientEmulator are test components at the integration testing level. They are compounded with the SUTs through the aggregation rules and interact with each other. They are the EPNs at level 2 and are encapsulated as EPAs of level 3. At level 3, TankWarSuiteTest represents a completed transaction between components. TABLE II below defines an aggregation rule for the level 3 action tankCanLeftMove.

**TABLE II**

<table>
<thead>
<tr>
<th>Element</th>
<th>Declarations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variables</td>
<td>TankServerEmulator $tse$; TankClientEmulator $tce0$; TankClientEmulator $tce1$; $int x$; $int y$; $int direction$; $String host$; $Socket sc$; $String message$; $boolean value$;</td>
</tr>
<tr>
<td>Event Types</td>
<td>$tanMove(int x, int y, int direction)$</td>
</tr>
<tr>
<td>Pattern</td>
<td>$tce0$.tanMove($int x$, $int y$, $int direction$)</td>
</tr>
<tr>
<td>Context Condition</td>
<td>$CanLeftMove$</td>
</tr>
<tr>
<td>Action</td>
<td>$create$ tankCanLeftMove($port$, $value$, $host$, $sc$, $message$, $x$, $y$, $direction$)</td>
</tr>
</tbody>
</table>

**B. Test Scenario**

In order to test whether the TankWar system executes the game properly, we realized the design scheme of the test framework for unit- and integration levels of testing. Firstly, we chose the TankServer class, TankClient class and ClientWar class as the SUTs at the unit testing level because they contain the basic and most important functions. And then we chose the TankServer class and TankClient class as the SUTs at the integration testing level for monitoring their interactions and detecting defects contained therein.

In order to check if the tester components work properly in testing the TankWar system, we implanted some errors in this system that may not be easily detected. In this test scenario, the tank’s movement was tested mainly. The testing is conducted under two different contexts: with and without employing the proposed test system.

Firstly, we assume that the game starts normally and test the tank’s movement without using the proposed test system. The `clientMove()` in ClientWar class moves the tank instance left by one place when the user presses the left arrow button. Now we made a change to the original source code of the left move method in ClientWar class so that the program has an error.

The program above runs properly in most cases. However, the tank instance’s position on the canvas can happen to exceed the left boundary or coincide with the coordinates of obstacles as a result of the left move. In that situation, it will cause an exception to be thrown in thread running the method and terminate the program. Since the `clientMove()` method above can be invoked properly in most cases, this kind of errors cannot be detected easily.

Lastly, we conducted the same tests on the same error-implanted program, this time with the proposed test system built on JUnit framework, as we did without the test system earlier. For example, we set the tank’s position to (200, 140), and ran the program as JUnit test. After testing, JUnit showed the results in the JUnit view window, as shown in Fig. 5. We can get the similar information from the test logger indicating that the test failed because the expected value is not equal to the actual value. Besides that, the TankWarSuiteTest also output the test results log to the Java Console.

![Fig. 5 Integration Testing in JUnit View](image)

The sequence diagram in Fig. 6 shows the sequence of interactions between the test components and the SUTs that take place when the testing is conducted according to the scenario stated above. In the diagram, the state symbol placed on a lifeline indicates the state or condition that must be true of the lifeline component after interactions have occurred up to that point where the state symbol is placed. In fact, when we start the test by JUnit framework, all of the methods are executed, one at a time.

The test system should behave properly in testing normal situations without malfunctions as well (i.e., it should not raise false alarms). Otherwise we cannot ensure that the test system is reliable and accurate in detecting errors of the target system. The test system displays test results of the seven methods as passed. In this case study, types of faults detectable are limited to those occurring when testing public methods. However, if instance variables of the target system were declared as private,
we could not have tested the methods properly because the test methods cannot access private variables.

V. CONCLUSION

In this study we proposed a CEP- and UTP-based testing method for testing event-based systems at unit- and integration levels. The test system built on JUnit testing framework according to this scheme increases the controllability and observability of event-based system testing without incurring changes or intervention to the source code of the target systems. By showing testing scenarios in which defects that might remain undiscovered were detected in our scheme, we demonstrated that the proposed scheme can increase the testability of software systems, in particular event-based systems. The increased testability can improve the efficiency of high-quality software development.

For future works, we plan to improve our test method in several ways. Some skeleton test code generation could be automated. Using the test components in conjunction with the JUnit framework, we can build a test system that automates much of the test process. In test process, we need some methods with which the software testers can prepare all the test data in advance, and when the test begins, the automated test system can iterate through all the test data with little human intervention. Then it can improve the efficiency of testing.

ACKNOWLEDGMENT

This research was supported by BK21 Plus Program.

REFERENCES


