World Academy of Science, Engineering and Technology
International Journal of Computer and Information Engineering
Vol:1, No:11, 2007

Automating the Testing of Object Behaviour: A Statechart-Driven Approach

Dong He Nam, Eric C. Mousset, and David C. Levy

***Abstract*—**The evolution of current modeling specifications gives rise to the problem of generating automated test cases from a variety of application tools. Past endeavours on behavioural testing of UML statecharts have not systematically leveraged the potential of existing graph theory for testing of objects. Therefore there exists a need for a simple, tool-independent, and effective method for automatic test generation.

An architecture, codenamed ACUTE-J (Automated stateChart Unit Testing Engine for Java), for automating the unit test generation process is presented. A sequential approach for converting UML statechart diagrams to JUnit test classes is described, with the application of existing graph theory. Research byproducts such as a universal XML Schema and API for statechart-driven testing are also proposed.

The result from a Java implementation of ACUTE-J is discussed in brief. The Chinese Postman algorithm is utilised as an illustration for a run-through of the ACUTE-J architecture.

***Keywords*—**Automated testing, model based testing, statechart testing, UML, unit testing.

I. Introduction

TESTING is the process of executing a program or a system for the purpose of improving the quality of the software [19]. Software testing is an integral and a necessary part of the software development life cycle, but it is also the most costly and time consuming task [18]. Therefore there is a need to form more intelligent tests and automate the procedure in order to improve the quality of the software being delivered.

Model Based Testing (MBT) is a methodology used for generating test cases based on the behavioural model of the system. In the past couple of decades Unified Modeling Language (UML) has become the de facto industry standard for modeling software systems as such [8].

Research in testing UML models has been around since the late 1990's, especially work involving generation of tests from class and statechart diagrams [10]. And work involving test generation and verification for Finite State Machine (FSM) date back even further.

Previous approaches include the use of UML for automatic test generation in which the model is compiled into an Intermediate Format (IF) [3]. Many approaches in statechart unit tests generation have been tool-specific [11, 14], addressing the problem using only vendor specific UML tools or with the use of independently developed applications.

Many of the past work do not address the problem of testing statecharts with multiple substate levels [12, 13]. Others have solved the problem by flattening the statechart to resemble FSM and applying common FSM testing techniques [10, 14]. But a common shortcoming in these approaches seems to neglect the range of existing traversal algorithms and their potential in achieving test objectives.

The objective of this research is to develop a tool-independent approach for automatic generation of unit tests for UML statecharts. The application of the approach to the JUnit framework, as well as the use of graph theory [4], is presented in this paper as an illustration of the concept.

The main dilemma in creating JUnit tests for implementation code is that there is no one standard for mapping statechart diagrams to Java code. A variety of known techniques are discussed in [2].

Most widely adopted approach is the use of nested switch statements. Scalability has been pointed out as a potential issue, especially in terms of readability and maintainability [7].

Another approach is to transform the statechart into an intermediate diagram called Testing Flow Graph, then generating test cases based on the test criteria [15].

Other approaches include the use of design patterns such as the State pattern [6], and State Table pattern [7] to describe statecharts. But since these patterns have primarily focused on encapsulating only the behaviour of the context state object, it is problematic when dealing with behaviour specifics and substates. Additionally, since design patterns are generic models for solving recurring problems, it does not describe in detail how to perform the direct mapping from models to Java code [9].

In recent studies, a new method has been devised for writing Java code based on UML statechart. The approach extends on the State design pattern and is based on object composition and delegation in order to solve the substate problem [1].

This paper assumes that the implementation of the statechart follows closely to the mapping technique described

Manuscript received October 25, 2005.

D. Nam is a postgraduate student at the School of Electrical and Information Engineering, University of Sydney, NSW 2006 Australia (phone: +61 2 9351 2337; fax: +61 2 9351 3847; e-mail: dnam@ee.usyd.edu.au).

E. C. Mousset is with the School of Electrical and Information Engineering, University of Sydney, NSW 2006 Australia (e-mail: mousset@ee.usyd.edu.au).

D. C. Levy is with the School of Electrical and Information Engineering, University of Sydney, NSW 2006 Australia (e-mail: dlevy@ee.usyd.edu.au).

in [1]. In grey-box testing, the test procedure has some knowledge of the implementation specifics of the system under test. This is necessary in the absence of structural information. And JUnit tests can then be generated for the appropriate class and methods based solely on the behavioural diagram such as statecharts.

II. State-of-the-Art: UML Modeling Tools

There are numerous commercial and open-source UML modeling tools available on the market today. They vary in functionality and price, as well as their support for different versions of UML and XMI. Following paragraphs give a brief account of two of the tools investigated.

IBM Rational Rose™ Enterprise Edition is one of the most widely used UML modeling tools in the industry and academia. Version 2003.06.13 supports UML 1.4 and has code generation capability in many different languages including Java. Rose Enterprise™ does not directly support XMI exporting, but an add-in developed by Unisys allows models to be exported in XMI 1.1.

Gentleware's Poseidon for UML™ Community Edition is a freely available tool for non-commercial use. It supports 9 of UML diagrams including the statechart diagram. Poseidon for UML™ is compliant with the UML 2.0 Diagram Interchange Standard and has limited Java code generation capability based on Class diagrams. The most useful feature of this tool however, is its support for statechart diagram subset of UML 2.0. And XMI 1.2 is used as the standard saving format for the models.

UML 2.0 is the latest and the current adopted specification. UML 2.0 Superstructure, which was completed in October 2004, is one of four parts in UML 2.0 specification. It describes thirteen structure, behaviour, and interaction diagrams that comprise UML. Although UML 2.0 is gradually replacing its predecessor UML 1.5, there is still a market for the previous version as vendors and open-source community battle to make the complete transition.

This last remark also applies to XMI 2.0 and is a key motivation behind our research and its materialisation as a testing tool. More specifically, a key design objective for the ACUTE-J architecture is to support a variety of versions of UML and XMI on a tool-independent basis.

III. System Architecture

ACUTE-J applies an MBT approach for automatically generating JUnit tests from UML statecharts. The automated process of ACUTE-J runs parallel to the development and implementation of the system. The architecture as depicted in Fig. 1 comprises of four main components: Semantic Formatter, Translator API, Test Generator, and Test Writer.

The test generation process begins with the modeling of the statechart using a UML tool which supports XMI exportation such as Rational Rose™. Once the statechart diagram has been generated and exported to XMI, it is passed onto Semantic Formatter. The key responsibility of Semantic Formatter is to produce an output XML file which contains only the statechart specific metadata. And with the use of Translator API, the XML document is parsed and stored as memory objects, ready for application of traversal algorithms.

There are many traversal algorithms which exist in graph theory [4, 5] that can be applied to a statechart testing. This paper looks at the Chinese Postman algorithm in detail and its application in testing statecharts with multiple substate levels. The penultimate stage of the statechart unit testing is carried out by Test Generator. It is responsible for generating test data which is then exported as an XML file for Test Writer to use in creating the final JUnit test class files.

.

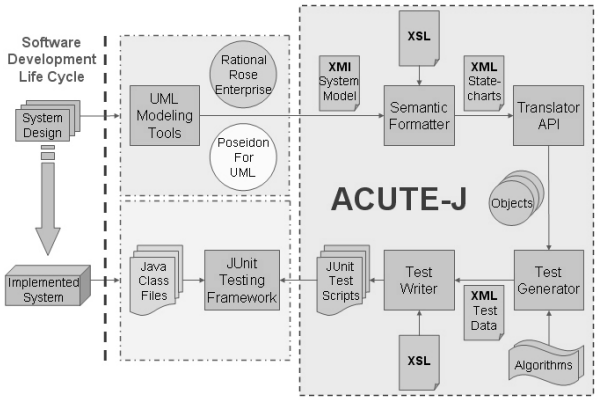

Fig. 1 ACUTE-J Architecture

A. Semantic Formatter

The Semantic Formatter plays two main roles. First is to achieve tool-independency and second, to achieve separation of concern.

The primary role of the Semantic Formatter is to move from a UML-centric representation, namely XMI, to a more universal, behaviour-oriented representation. The Semantic Formatter as the name suggests is responsible in catering for the varying differences of UML tools available to the modelers, and the many flavours/versions of XMI that are supported by these tools. There is a need for a continual support of older and obsolete versions until the most recent XMI 2.x is fully implemented by all UML tools.

A typical XMI file exported using a UML tool shows statechart information nested amongst large amount of information regarding other UML diagrams and graphical formatting specifics. The secondary functionality of the Semantic Formatter is to filter out this irrelevant information and produce a workable XML file containing only the necessary information regarding the states, transitions, guards, and signals.

A behaviour-oriented representation, in the form of an XML Schema and as a target space for the Semantic Formatter is currently under study.

B. Translator Object Model and API

Another by-product of our research is a behaviour-oriented object model and API for the in-memory representation of the statechart.

Many of the previous approaches [14, 17] make use of XSLT alone for producing test cases, but when complex graph algorithms requiring graph Eulerization and traversal path determination; it makes sense to provide an API to support these tasks. The functional role of the Translator is to package the statechart information in a way allowing efficient application of searching and traversal through the statecharts.

The Translator API is also meant to be versatile, allowing users to define new algorithms, or to plug-in existing algorithms of their choice.

C. Algorithms

The role of the algorithms is to determine the path in which the statechart is traversed for the purpose of testing.

There are limitless numbers of existing graph algorithms which can be applied to determine the test path such as shortest round trip, depth first search, most likely paths, etc. One of the simplest ways of coverage testing is to form a random path through the statechart. The random path algorithm can be used for exhaustive stress testing of the test object. However, because there is no guarantee that all states and transitions will be covered, it is not meaningful in the larger scheme of coverage testing.

One of the more effective and efficient graph algorithms used to satisfy coverage is the Chinese Postman algorithms. The Chinese Postman algorithm, discovered by a Chinese mathematician Kwan Mei-Ko in 1962, is based on the problem of delivering mail along one-way streets in the quickest and most efficient manner with least number of travels along the same streets. This problem is easily adapted to statecharts. The transitions represent the one-way streets that must be traversed, and states are nodes or intersections that join the streets.

The expected benefits of applying the Chinese Postman algorithm is that the graph component of the statechart can be visited in an efficient manner with the minimum number of transitions whilst guaranteeing total transitional coverage. Thus ensuring that every transition represented by method calls is tested, and every possible state of the context object is validated.

Unlike the conventional method of testing via flattening of composite states, this research explores a new approach to testing of statecharts with multiple substate levels. In our approach, each level of the statechart is handled as a separate distinct problem as shown in Fig. 2. Each statechart are then Eulerized independently, Fig. 3, and recomposed into one statechart solution with a complete path, as shown in Fig. 4.

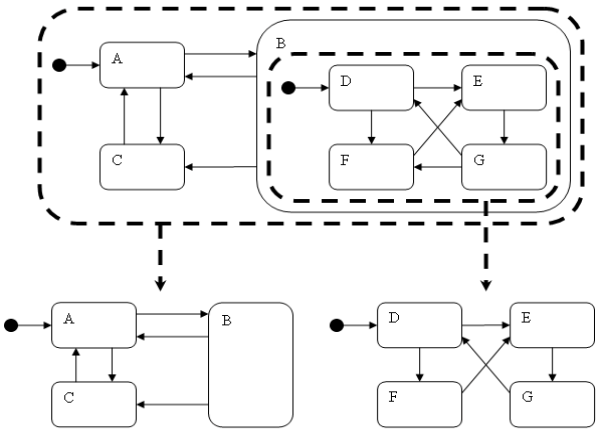

Fig. 2 Problem Division

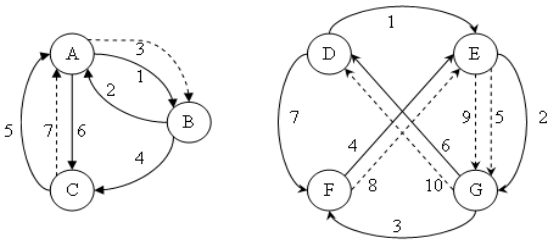

Fig. 3 Eulerized Statecharts

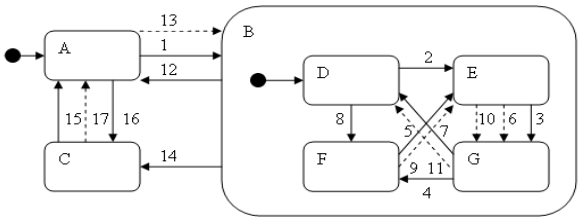

Fig. 4 Recomposed Statechart Solution

D. Test Generator

Test Generator is responsible for producing the body of the tests. This is achieved by applying the algorithms such as the aforementioned against the statechart objects in memory. The test data is constructed by generating the expected results of the state variables according to the model and according to the traversal path undertaken by the algorithm.

There are two broad types of tests that are generated for a given statechart: navigational and behavioural. Testing of navigation involves not only ensuring that all states are reachable and all transitions can be made by the object, but also guaranteeing that the context object remains in a valid state at all times (see “Section IV. Results” for an illustration).

Behavioural testing is concerned with testing the guards of transitions. This ensures that a transition to a new state is only

made if the guard condition is satisfied. An example of such validation includes checking the upper and lower limits of an integer variable, the format of character string, etc.

Test Generator outputs a single XML file containing information regarding the package, classes, and methods to be tested, as well as the body of the unit test.

E. Test Writer

The role of Test Writer is simply to apply XSLT 2.0 on the test data and produce multiple JUnit test class files which can be executed under the JUnit Testing Framework. Test Writer can generate a complete suite of tests according to the graph algorithms used by Test Generator to achieve different test objectives.

IV. Results

This section will cover a brief example of the application of ACUTE-J to generate transition coverage test using the Chinese Postman algorithm.

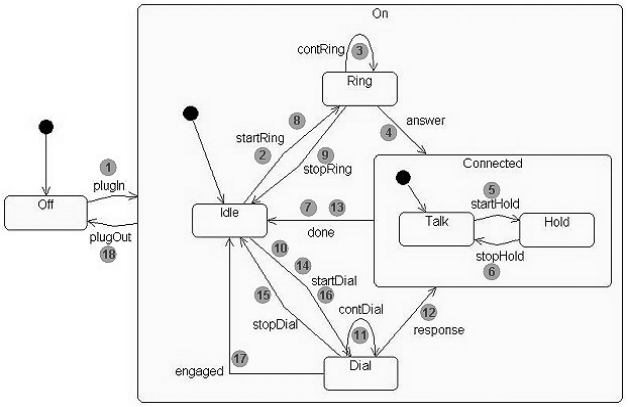

Fig. 5 Statechart Diagram of a Phone

The diagram depicted above in Fig. 5 represents a simple model of a typical house phone. The phone can be either Off or On, and its substate behaviour in the On state consists of the Idle, Ring, Dial, and Connected states. And for the purpose of this example the Connected state also contains substates Talk and Hold.

The statechart diagram may seem simple enough to understand its behaviour. But in order to test the transitional paths, what is the quickest and most efficient path to take in order to test every transition and visit every state? This problem cannot be readily addressed by mental application, especially as statechart becomes large and the substate levels grow.

The result of ACUTE-J's application of the Chinese Postman algorithm can also be seen in Fig. 5. The path identified to be the most efficient is shown by the numbers in the circles. As a result, all transitions in the statechart are traversed and all states including substates are visited at least once.

Once an algorithm has been applied by Test Generator, Test Writer produces the JUnit test class for the Phone context object. Listing 1 on the following page shows part of the final TestPhone.java class. The commented numbers beside the method calls corresponds to the path steps identified by the application of the Chinese Postman algorithm. These numbers are for illustration purposes only and are not part of the generated test code by Test Writer.

The body of the test can be seen in the testSetState() method. Important things to note are phone0.state which holds the current state of the context object Phone; also, method calls such as phone0.plugIn() correspond to the transitions in the statechart.

```
1  import junit.framework.*;
2  import junit.textui.TestRunner;
3
4  public class TestPhone extends TestCase {
5    Phone phone0;
6
7    public TestPhone(String name) {  super(name); }
8
9    protected void setUp() { phone0 = new Phone(); }
10
11   protected void tearUp() { phone0 = null; }
12
13   public void testSetState() {
14     assertTrue(phone0.offState.equals(phone0.state));
15     phone0.plugIn();                                              // 1
16     assertTrue(phone0.onState.equals(phone0.state));
17     assertTrue(phone0.idleState.equals(phone0.state.state));
18     phone0.startRing();                                           // 2
19     assertTrue(phone0.onState.equals(phone0.state));
20     assertTrue(phone0.ringState.equals(phone0.state.state));
21     phone0.contRing();                                            // 3
22     assertTrue(phone0.onState.equals(phone0.state));
23     assertTrue(phone0.ringState.equals(phone0.state.state));
24     phone0.answer();                                              // 4
25     assertTrue(phone0.onState.equals(phone0.state));
26     assertTrue(phone0.connectedState.equals(phone0.state.state));
27     assertTrue(phone0.talkState.equals(phone0.state.state.state));
       .
       .
       .
28     phone0.startDial();                                           // 16
29     assertTrue(phone0.onState.equals(phone0.state));
30     assertTrue(phone0.dialState.equals(phone0.state.state));
31     phone0.engaged();                                             // 17
32     assertTrue(phone0.onState.equals(phone0.state));
33     assertTrue(phone0.idleState.equals(phone0.state.state));
34     phone0.plugOut();                                             // 18
35     assertTrue(phone0.offState.equals(phone0.state));
36   }
37
38   public static void main(String[] args) {
39     TestRunner.run(TestPhone.class);
40   }
41 }
```

Listing 1 TestPhone.java

The test begins by checking that the initial state of the object is Off, as highlighted by line number 14 in the listing. Once the first traversal phone0.plugIn() is made, the new

current state of the context object is verified. Line 16 checks for the On state and line 17 checks for the Idle substate. The test process continues in this manner until the entire path has been validated. At each step, superstates and substates of the Phone are checked. Lastly we ensure that the Phone object is again in Off state, as line 35 shows.

V. Discussion

The architecture of ACUTE-J described in this paper is part of a continuing work in automated statechart unit testing.

The first and most obvious question that may arise is the use of ACUTE-J specific XML schema and object model. While the justification for such approach is to improve the accessibility and modifiability of the statechart for algorithms, it brings an opportunity to propose to the community; an invitation for reflection on the matter, namely capturing and representing responsibility-oriented information.

The advantages brought on by the use of the object model and the API is significant. It opens up an opportunity to apply complex graph algorithms for testing. These algorithms require graph manipulations such as transition duplication, and path determination which is not possible with using XSLT.

Navigational testing or transition coverage testing described in the results section contain minor limitations. Consider a top level state within a statechart without any outgoing transitions. Such object will remain in the transition-less state until the termination of the object. Current implementation of Chinese Postman algorithm does not deal with this special case. Perhaps this kind of problem should be considered as a defect within the model and handled syntactically by UML tools.

One way such models can still be tested is to devise look-ahead algorithms purposed to detect transition-less states and clone the testing path before making the final transition into state-of-no-return. This way, these types of states can still be tested in separate test cases, without affecting the rest of the testing paths, and having to retrace through the statechart.

On a similar note, the testing of isolated states within a statechart diagram is not accounted for in the present version of the ACUTE-J architecture, for it is impossible to reach such states via valid transition. The current algorithm will simply ignore this kind of flaw in the model.

ACUTE-J test generation regime currently deals only with simple statechart models. Complex features of UML statechart diagram including history pseudostates, state reactions, deferred events, synchronisation table, and orthogonal regions fall outside the functional specification of ACUTE-J.

Finally, the testing approach described in this paper can easily be extended to the testing of behaviour of components, web services and their combination as workflows; where the validation of behaviour and responsibility is the key.

VI. Conclusion

The key benefits of applying the method as described in this paper are the increase in the quality of software by introducing behaviour-relevant validation at the level of the unit testing phase. This approach also encompasses leveraging graph theory and related algorithms for achieving the test objective, as highlighted in this paper with the use of Chinese Postman algorithm.

Lastly, it opens an invitation for the community to reflect on an XML schema for the packaging of state and responsibility of objects/components.

References

[1] I. A. Niaz and J. Tanaka, "An Object-Oriented Approach To Generate Java Code From UML Statecharts," *International Journal of Computer & Information Sceinces (IJCIS)*, vol. Vol. 6, 2005.

[2] I. A. Niaz and J. Tanaka, "Mapping Statecharts to Java Code", *IASTED International Conference on Software Engineering (SE2004)*, Innsbruck, Austria, February 2004.

[3] C. Crichton, A. Cavarra, and J. Davies, "Using UML for Automatic Test Generation", *Automated Software Engineering*, 2001.

[4] H. Robinson, "Graph Theory Techniques in Model-Based Testing", *International Conference on Testing Computer Software*, 1999.

[5] H. Robinson, "Intelligent Test Automation". *Soft-ware Testing & Quality Engineering magazine,* September-October 2000. http://www.geocities.com-/harry_robinson_testing/robinson.pdf

[6] E. Gamma, R. Helm, R. Johnson and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1996.

[7] B.P. Douglass, *Real Time UML – Developing efficient objects for embedded systems*, Addison-Wesley, Massachusetts, 1998.

[8] I. K. El-Far and J. A. Whittaker, "Model-based Software Testing," Encyclopedia on Software Engineering, 2001.

[9] P. Metz, J O'Brien and W. Weber, "Code Generation Concepts for Statechart Diagrams of the UML v1.1", *Object Technology Group (OTG) Conference*, University of Vienna, Austria, June 1999.

[10] Y.G. Kim, H.S. Hong, S.M. Cho, D.H. Bae, S.D. Cha, "Test Cases Generation from UML State Diagrams", *IEE Proceedings – Software*, 146(4):187–192, August 1999.

[11] J. Offutt and A. Abdurazik, "Generating Tests from UML Specifications", *Second International Conference on the Unified Modeling Language (UML 99)*, Fort Collins, CO, October 1999, pp. 416-429.

[12] J. Hartmann, C. Imoberdorf, and M. Meisinger, "UML-Based Integration Testing", *International Symposium on Software Testing and Analysis 2000 (ISSTA 2000),* Portland, USA, 2000.

[13] J. Takahashi and Y. Kakuda, "Extended-model Based Testing by Directed Chinese Postman Algorithm", *7th IEEE International Symposium on High Assurance Systems Engineering (HASE'02),* Tokyo, Japan, 2002.

[14] J. Grundy, Y. Cai and A. Lui, "Generation of Distributed System Test-beds from High-level Software Architecture Descriptions", *Automated Software Engineering Conference*, San Diego, USA, 2001.

[15] Li. Liuying and Q. Zhichang, "Test Selection from UML Statecharts", *31st International Conference on Technology of Object-Oriented Language and Systems*, Nanjing, China, 1999.

[16] S. Kansomkeat and W. Rivepiboon, "Automated-Generating Test Case Using UML Statechart Diagrams", *2003 Annual Research Conference of the South African Institute of Computer Scientists and Information Technologists on Enablement Through Technology*, South Africa, 2003.

[17] M. J. Rutherford and A.L. Wolf, "A Case for Test-Code Generation in Model-Driven Systems", *Second International Conference on Generative Programming and Component Engineering,* Erfurt, Germany, 2003.

[18] N. Eickelmann and A. Willey, "An Integrated System Test Environment", *14th International Internet & Software Quality Week 2001 (QW2001)*, San Francisco, May 2001.

[19] Testing Software Based Systems: The Final Frontier [Online]. Available: http://www.softwaretechnews.com/stn3-3/final-frontier.html