UML STATECHART & ALGEBRAIC SPECIFICATION BASED CLASS LEVEL TESTING IN SOFTWARE ENGINEEING

AMIYA TRIPATHY
Lecturer, Don Bosco Institute of Technology,
Kurla (W), Mumbai, India.
amiya@donboscoit.ac.in

ADITYA SINHA
Lecturer, National Institute of Technology,
Jameshedpur, Jaharkhandh, India.
adityaksinha@yahoo.com

Abstract. There is an increasing need for effective, fast and automatic testing of software. Tests are commonly generated from program source code, graphical models of software and specification / requirements. UML is increasingly being used for software design. UML diagrams also provide a significant opportunity for testing because they precisely describe how the functions provided by software are connected in a form that can be easily manipulated by automated means. We present a methodology for class level automatic testing of .NET components using UML statechart and algebraic specification. It checks for the consistency between software specification and .NET implementation. This is a useful framework for testing any software that exhibits state machine intensive behavior. Test oracles are automatically derived through axioms provided in algebraic specification. We implemented a prototype of our methodology, which is able to automatically generate test cases, test data and test oracle. It automatically creates a test driver. Generated test driver checks for the consistency between specification and implementation and gives test report. This paper presents prototype of our methodology with case study.

1 Introduction

It is estimated that testing often accounts up to fifty percent of the overall software costs. A large amount of time and money within the test process may be attributed to incomplete, inconsistent or ambiguous informal specifications of test objects. Software companies have been able to improve upon several dimensions of software quality and reduce development time. However, no significant achievement has been made in controlling the growth of bugs and defects. According to a recent article, defective code remains the hobgoblin of the software industry, accounting for as much as 45% of computer system downtime and costing U.S. companies about 100 billion last year in lost productivity and repairs. Software testing is an indispensable tool in moving toward the goal of greater software quality, and often consumes a great deal of development resources in practice. A more formal approach to the early phases of software development can reduce the error rate drastically and, in addition, can significantly improve the central testing activities like test case Design and test Evaluation.

There are many ways of testing. One way is manually running all tests from the keyboard i.e. Hands-on testing. It is common throughout the industry today because it provides immediate benefits, but in the long run it is tedious for the tester and expensive for the organization. Another way is static test automation. The static automation scripts exercise the same sequence of commands in the same order every time. These scripts are costly to maintain when the application changes. The tests are repeatable; but since they always perform the same commands, they rarely find new bugs. A different approach is using random test programs. They come up with unusual test action sequences and find many crashing bugs, but it's hard to direct them to the specific parts of the application that we want tested. Since such tester does not know what they are doing, they miss obvious failures in the application. Best approach is to combine other tester's approaches with a type of intelligent test automation called model-based testing. The model-based testing doesn't record test sequences verbatim like static test automation does, nor does it bang away at the keyboard blindly. The model-based tests use a description of the applications behavior to determine what actions are possible and what outcome is expected. This automation generates new test sequences endlessly, adapts well to changes in the application, can be run on many machines at once, and can run day and night.

Large systems are inherently complex to test and require, regardless of the test strategy, large numbers of test cases. If a system testing method requires the tester to perform frequent, complex manual tasks, then such a method is not likely to be usable in a context where time to market is tight and qualified personnel is scarce. Moreover testing is perceived as tedious, uncreative, boring work by practitioners. Therefore, the potential for automation of a test methodology is an important criterion to consider.

Because of the growing importance of object-oriented programming, a number of testing strategies have been proposed. Object-oriented programming consists of several different levels of abstraction; namely the algorithmic level,

class level, cluster level, and system level. The testing of object-oriented software at the algorithmic and system levels is similar to conventional programming testing. Testing at the class and cluster levels poses new challenges. Since methods and objects may interact with one another with unforeseen combinations and invocations, they are much more complex to simulate and test than the hierarchy of functional calls in conventional programs. In this report a new methodology for object-oriented software testing at the class levels is discussed.

For class-level testing, it is it is necessary to check for the consistency between specification and implementation. To address this issue we are using UML statechart diagrams with algebraic specifications. Fundamental pairs of equivalent ground terms are embedded in UML statechart diagrams. This methodology is able to generate test cases with test oracle and test driver. We are using a black-box technique and then it is determined whether the objects resulting from executing such test cases are observationally equivalent. We have tested this methodology with .NET components, yet same methodology can be used any other type of components as well.

1.1 Related work

Existing works [10, 3, 2, 12, 11, 7] discuss how to automatically generate test cases (transition sequences and input data) for achieving certain coverage criteria. However, the proposed approaches have the following limitations. First, the UML statechart notation is not fully supported. State diagrams used in [10, 3, 7] are not UML statecharts, though they correspond to UML statecharts with very simple features. For instance, state diagrams in these works can be regarded as statecharts that involve call or signal events, with guards but without actions. Among the remaining three works, event type in [12] is not specified whereas only change events [11] are accounted. Although the UML defines seven different types of actions, actions are not taken into account except in [11, 12, 2], where actions are either assignment (to attributes), return, or call action. Last, guards are not considered in [11]. These are important limitations as they limit the applicability of the different approaches. Second, in defining operations, only simple modifications are supported by events and actions, namely, modifications that are simple enough to be captured in the form of assignments to attributes. More importantly, existing works do not suit object oriented System well. For example, associations among classes cannot be represented in existing works and this might explain why existing works considered statechart in isolation without accounting for the interactions among object statecharts.

Chen, Tse and Chen [4] present a new methodology for testing called TACCLE (Testing at the Class and Cluster Levels). They have an interesting insight into object oriented testing. The research details a method for testing the interactions between related objects properties and methods, what they call a cluster. A specification language called Contract is used to describe the interactions between objects. The article describes testing using fundamental equivalent and non equivalent terms.

1.2. Objective

Our objective is to build a test automation framework that works by receiving a UML Statechart and algebraic specification of the component to be tested and .NET executable version of the component. We intend to determine set of test cases along with test oracle and a test driver to automatically run these test cases.

2 Background Information

We first provide necessary background information about this report, clarify relevant issues, and state the assumptions we make in this report.

2.1 UML

This section is a brief introduction to UML. For complete references, see [9]. UML is a third generation semi-formal modeling language used for specifying, visualizing, constructing, and documenting object-oriented systems [8]. UML rigorously defines the semantics of the object meta model and provides a notation for capturing and communicating object structure and behavior. It is the de facto industrial standard developed, maintained, and managed by the Object Management Group (OMG) including many methodologies - including Grady Booch (Booch Method), Jim Rambaugh (Object Modeling Technique[OMT]), Ivar Jacobson (Object-oriented software engineering[OOSE]), and David Harel (Statecharts). UML is not a programming language. We cannot write a program in UML. Though some CASE tools provide code generation capabilities from UML models, the generated code is nothing more than a framework. Developers still need to write code to implement the methods.

2.2 XMI

Object Management Group (OMG) has defined XMI [5] standard for data and metadata interchange. It enables exchange of any kind of metadata that can be expressed by using the MOF (Meta-Object Facility)[6]. It integrates three industry standards MOF, UML and XML [1]. The fact that OMG defines the UML meta-model as a MOF meta-model

implies that XMI serve as an interchange format for UML. OMG has defined both a Document Type Definition (DTD) and XML Schema [1] for XMI documents. To distinguish between XMI in general and its application to UML, the format is often referred to as XMI[UML]. XMI employs XML as the encoding format. XMI specifies an open information interchange model that is intended to give developers working with object technologies, the ability to exchange programming data. It is used to exchange data and metadata between different tools. XMI provides many advantages: XMI document can be validated against a DTD or an XML Schema, an XMI document can employ Xlink technology to address elements in other documents, it is extensible, i.e., tools are permitted to extend the basic elements, and XMI can transfer the difference of the documents so that the overall documents needs only to be transferred once. However the disadvantages are that there are too many versions of XMI (version 1.1, 1.2, 2.0), XMI does not define any standards for pictorial rendition of contained data, and the XMI files are generally very large in size. There are many commercial and open source UML modeling tools, which support XMI. A plug-in from UNISYS, for Rational Rose 2002 [2], is used to export XMI generated for UML Statechart diagram. We have not tested the differences among various other UML modeling tools in interpreting the generated XMI. The UNISYS plug-in supports XMI version 1.0 and 1.1. We use XMI version 1.1 and UML version 1.3 for decoding UML Statechart diagrams.

2.3 .NET ASSEMBLY

When we build or programs or components in .NET, both metadata and code are stored in self-contained units called assemblies. An assembly stores information about the components and resources against which it is compiled, as well as the types defined within it. The Common Language Runtime (CLR) accesses this information; we can access this metadata information as well. To use assembly we must use the System.Reflection namespace, which is where the Assembly class is defined. There are many info Classes within the System. Reflection namespace i.e. MemberInfo, MethodInfo, ParameterInfo, ConstructorInfo, FieldInfo and EventInfo. Using all these info classes complete detail of component can be derived. We are using .NET assembly of component to create object of the classes for which class level testing to be done.

3 STATECHART DIAGRAM

Harel statecharts form the basis of UML statecharts. Statecharts overcome the limitations of traditional FSMs while retaining benefits of finite state modeling. Statecharts include the notions of both nested hierarchical states and concurrency while extending the notion of actions. Statecharts consist of states, transitions, synch states, and a variety of different state like things called pseudo states.

According to [9],"State machines can be used to specify behavior of various elements that are being modeled". Statechart diagrams are the diagrammatic representation of state machines. Statecharts can be used to describe the behavior of individual entities (e.g., a class) as well as a collection of entities (e.g., class cluster, subsystem, system etc.). From now on, this report will assume class statecharts only. Nevertheless, conclusions and results obtained from class statecharts can easily be generalized to include other types of statecharts.

One important issue is the one of testability: The degree to which a model (in our case, a UML Statechart model) has sufficient information to allow automatic generation of test cases [5]. Since the use of the UML notation is not constrained by any particular, precise method, one can find a great variability in terms of the content and form of UML artifacts, whether at the analysis or design stages. However, the way UML is used determines the testability of the produced UML artifacts. We therefore address the testability requirements we need to impose on UML artifacts. In the following, we'll present testability issue of statechart with definitions.

3.1 ASSUMPTIONS WITH STATECHART

A state machine consists of a number of transitions and states. A transition consists of an event, a guard condition, and an action sequence. The following definitions are taken from [8]. A transition specifies a directed relation between two states that when the specified event occurs and the specified guard is satisfied, the object will leave the first state, perform specified actions, and enter the second state. An event models external stimulus (input) to the state machine; a guard is a Boolean expression evaluated when an event occurs; an action models the response (output) of the state machine. A special type of action is the activity, which is an ongoing task that an object executes while it stays in a specific state.

Next we discuss the testability of UML statecharts.

3.2 STATECHART FLATTENING

It is common practice to model the complex behavior of an object using composite state and concurrent sub-states. The use of such mechanisms helps to cope with complexity but makes it hard to generate test cases. It is not obvious how many distinct states and transitions are there in the statechart. In order to apply the state-based criteria mentioned in the introduction, it is necessary to remove all hierarchy and concurrency in the statecharts and obtain flat statecharts, in which every distinct state is represented by a

node and all possible transitions are shown explicitly [2]. The transformation from hierarchical and concurrent statecharts into flat statecharts is explained in [12] and in this report we assume there are such algorithms to flatten user-supplied statecharts.

3.3 NONDETERMINISM

The semantics of UML statecharts allow for the possibility of non-determinism in state transitions: Multiple transitions may be enabled within a state machine [9]. For example, when two transitions originating from the same state are enabled by the same event, then only one of them will fire. If firing priorities are not specified, the selection of which transition to fire is non-deterministic. This research does not handle such cases as we assume the statecharts deterministic.

3.4 EVENT

According to the UML specification [UML01], there are five types of events: call, signal, change, time and completion1 event. We do not account for the completion event because it represents the completion of an activity rather than an explicit event [9]. Therefore its occurrence depends on when an activity completes its execution. In our state-based testing environment, we want the test driver to have full control over when transitions fire and this can only be done by creating explicit events to trigger transitions.

In the UML meta model, only a call event is associated with an operation [9], which is called the event handler. A call event is more complex than other types of events since its event handler can modify the collective state while other types of events only affect the state of one object. In this work, only the effect of call events is accounted for. The effect of other types of events is ignored. The effect of a call event is captured by the post condition of its event handler.

3.5. Action

Eight types of actions are defined in the UML specification [9]: assignment, call, create, destroy, return, send, terminate, and uninterrupted.

A return action returns the control to the object that invokes its execution and this has no effect on the collective states. The terminate action causes the self-destruction of the owning object of the state machine. Because the owning object will no longer exist after its execution, the terminate action is not interesting for this work and is thus not considered.

This report considers the following actions, which might change the collective state: assignment, call, send, create, and destroy. Among these types of actions only the call action is associated with an operation and therefore has its effect explicitly modeled in the operation.

In this report it is desirable that the effects of all types of actions are captured by algebraic specifications. We are using stereotype facility of UML to put algebraic specification in UML statechart. Ground terms from this algebraic specification are extracted and testcases with test oracle is prepared using our XMI Parsing mechanism.

4 Algebraic Specifications

An algebraic specification for a class is composed of a syntax declaration and a semantic specification. The syntax declaration lists the operations involved, as well as their domains and co-domains, corresponding to the input parameters and output of the operations. The semantic specification consists of axioms in the form of conditional equations that describe the behavioral properties of the operations.

An algebraic specification for a class is composed of a syntax declaration and a semantic specification. The syntax declaration lists the operations involved, as well as their domains and co-domains, corresponding to the input parameters and output of the operations. The semantic specification consists of axioms in the form of conditional equations that describe the behavioral properties of the operations.

4.1 Terminology and notations used in Algebraic specification

Before discussing about class level testing using UML statechart and algebraic specification. let's have a look on basic terminology and notations of algebraic specification. A **term** is a sequence of operations in an algebraic specification. For example,

new.push(10).push(20).pop

is a term in the class of integer stacks above. A term without variables is called a **ground term**.

If a subterm within a ground term is unified against the left-hand side of an equational axiom and substituted by the right-hand side of the axiom, it means that the ground term is transformed into another using the axiom as progressive left to- right rewriting rules. A ground term is in **normal form** if and only if it cannot be further transformed by any axiom in the specification. For example, *new.push(10).push(20)* is in normal form, but *new.push(10).push(20).pop* is not, since the latter can be transformed by axiom *a4* into *new.push(10)*.

An algebraic specification is said to be **canonical** if and only if every sequence of rewrites on the same ground term reaches a unique normal form in a finite number of steps. In a given class C, operations or methods that return the values of the attributes of the objects in C are called the **observers** of C. Operations or methods that return initial objects of C are called **creators** of C. Operations or methods that transform the states of objects in C are called **constructors** or **transformers** of C. The current state of an object is the combination of current values of all attributes of this object. When a constructor or transformer acts on an object, it changes the value of at least one attribute of the object. The difference between a constructor and a transformer is that a transformer may be eliminated from a term by applying rewriting rules, but a constructor may not. In Example 1, for instance, the operation *new* is a creator, *push(N)* is a constructor, *pop* is a transformer, and *is Empty* and *top* are observers.

An **observable context** on a class C is a sequence of constructors or transformers of C (possibly an empty sequence) followed by an observer of C. For example, *push(100).push(200).pop.top* is an observable context on the class Stack. The observer *top* is also regarded as an observable context on Stack.

Definition 1 *(Observational Equivalence of Object).* Given a canonical specification and an implementation of a class C, two objects O1 and O2 are said to be **observationally equivalent** (denoted by O1 $\approx_{obs}$ O2) if and only if the following condition is satisfied:

If no observable context *oc* on C is applicable to O_1 and O_2, then O_1 and O_2 are identical objects. Otherwise, for any such *oc* on C, O_1 *oc* and O_2 *oc* are observationally equivalent objects.

5 Testing Methodology

The testing process of our methodology begins with the tester indicating the .NET component to be tested and the statechart with algebraic specification for the class behavior. Based upon this information, the tester defines the representation mapping by associating the code to the specification. Based upon these selections, our framework automatically produces test drivers (with embedded test cases and test oracles) to satisfy the criterion and compiles/executes the test drivers with user given parameters. At the end of this process, failures, detected by the test oracles as discrepancies between behavior and the statechart specification, are presented to the user. Fig. 1 illustrates this process. In the follow sections, we further explain each step in this process, highlighting our efforts.

5.1 Test Driver generator

Test driver generator is the heart of whole mechanism. Through the help of XMI Parser it reads XMI representation. Using .NET it reads executable version of the implemented component and based on these two information's it creates a test driver in C#. With the help of XMI Parser it writes transition table.

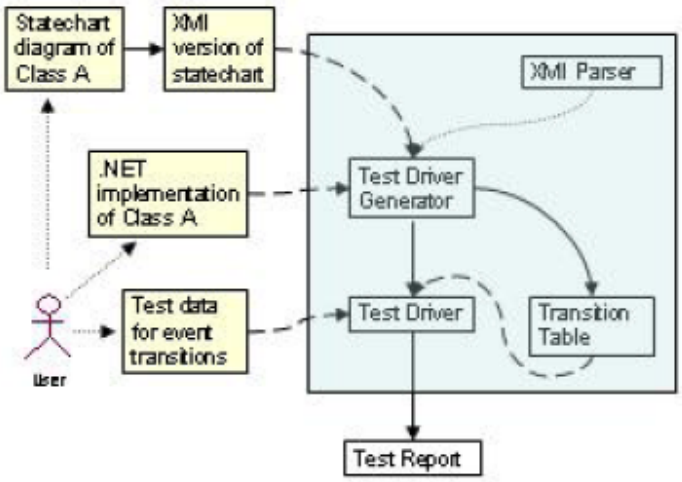

Fig.1. Flow of automatic testing

5.2 Test Driver

Test driver generator creates test driver. When Test driver is executed first it creates an object of the class to be tested using class constructors and puts object in an initial state (.NET provides support to create objects of non system scope classes), initial state information is provided in transition table. Test driver reads every transition of transition table and receives test data from the data domain provided with event argument. After the execution of triggering event test driver checks for the consistency between component implementation in .NET and component specification in UML statechart using test oracles which are embedded in transition table. Using algebraic specification equivalence of transitions is identified which helps to determine finite number of testcases.

6 Conclusions

Our Method that is UML Statechart and algebraic specification based automatic software testing method attempts to verify if the implementation's behavior is consistent with the statechart and collaboration based specification, detecting failures of the implementation. It does so by generating a test driver.

This test driver goes through various sequences of states (based upon the specification, stored in transition table) of an object and verifies correctness of the object's attribute values as it transitions between states and whether the transitions are to the correct state, thereby meeting the specified behavior.

References

[1] http://w3.org/TR/REC-xml. *Extensible markup language (xml)*.

[2] R. V. Binder. *Testing Object-Oriented Systems, Models, Patterns, And Tools.* Addison Wesley Longman, Inc., 2000.

[3] N. Parrington I.Mitchell B.Y.Tsai, S.Stobart. An automatic test case generator derived from state-based testing.

[4] Huo Yan Chen, T. H. Tse, and T. Y. Chen. Taccle: a methodology for object-oriented software testing at the class and cluster levels. *ACM Trans. Softw. Eng. Methodol.*, 10(1):56–109, 2001.

[5] Object Management Group. *XML Metadata Interchange (XMI)*. Technical report, Object Management Group., http://www.omg.org.

[6] Object Management Group. *Meta Object Facility (MOF)*. Technical report, Object Management Group, specification (version 1.3). Edition, March 2000.

[7] A. Abdurazik, P. Ammann J. Offutt, and S. Liu. Generating test data from state-based specifications.

[8] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language Reference Manual*. Addison Wesley Longman, Inc., 1999.

[9] G. Booch, J. Rumbaugh and I. Jacobson. *Unified Modeling Language (UML)*. OMG, 2001.

[10] G.H.Travassos and M.E.R.Vieira. Technology of object-oriented languages and systems.

[11] A. J. Offutt and A. Abdurazik. Generating tests from uml specifications.

[12] S. M. Cho, D. H. Bae, S. D. Cha, Y. G. Kim and H. S. Hong. Test case generation from UML state diagrams.