Software Testing and the UML

Clay E. Williams
Center for Software Engineering, IBM T. J. Watson Research Center
clayw@us.ibm.com

1. Introduction

The Unified Modeling Language (UML) has emerged as an industrial standard for modeling software systems, and has been presented to the International Organization for Standardization for consideration as an international standard [4]. UML has received a great deal of attention (both positive and negative) from the software design and development communities, and work is ongoing to enhance and expand its capabilities. However, the software testing community has had much less awareness and debate about UML, and has largely been absent as the modeling standard was developed. This is an important issues, because in many software development organizations, the cost of testing can account for more than 40% of the total development cost for a software system. Given these facts, this abstract seeks to explore the possibility of using the UML for software testing.

2. The UML

The UML is a visual modeling language that can be used to "specify, visualize, construct, and document the artifacts of a software system" [7]. The language itself is specified using a 4-layer metamodel architecture, which partitions the UML into three logical subpackages: Foundation, Behavioral Elements, and Model Management. UML provides seven views into a software system. These are the static, use case, state machine, activity, interaction, physical, and model management views [7].

The Foundation package provides the basic infrastructure for exploring the static structure of systems. This infrastructure includes:

- class diagrams - class structure and associations
- component diagrams- map classes to software units
- deployment diagrams - physical system structure

The Behavioral Elements package provides the linguistic elements for modeling the dynamic behavior of the system and includes:

- use case diagrams - functionality as seen by actors
- interaction diagrams - object message sequences
- activity diagrams - business / software processes
- state diagrams - object state transition behavior

In seeking to explore the issues that arise when using UML in the software testing process, we will focus primarily on the diagrams in the Behavioral Elements package. This is because most of the activities in software testing seek to discover defects that arise during the *execution* of a software system, and these defects are generally dynamic (behavioral) in nature [3]. However, there are cases where the behavioral information will need to be augmented with static information typically found in the Foundation package.

Taken together, the various UML diagram types provide the capability to explore the static structure, dynamic behavior, and physical deployment of a software system.

3. Model Based Software Testing

UML models are typically developed and used when building object-oriented (OO) software systems. An extensive body of literature exists to address the issues associated with testing OO systems. I present only a cursory summary of the field here. For more information see one of [3,5,6,8].

The vast majority of the work examining model based testing of OO systems focuses on the use of either class or state diagrams. Class diagrams provide information about the public interface of classes, method signatures, and important relationships between classes. State diagrams provide information about the behavior of a class (or set of classes.) The UML had a predecessor modeling method known as the Object Modeling Technique (OMT) that added a functional model, which was also used for testing [6].

The Unified Process, which was developed by Ivar Jacobson and others at Rational Software (where the UML was initially developed as well) indicates that the primary diagrams of interest to testers will be use case diagrams and their corresponding realizations as interaction (sequence or collaboration) diagrams [2].

These previous approaches ignore much of the complexity of testing real world systems, such as the fault models that apply to the system as it moves through the test phases [1]. In the section below, I outline an agenda for applying the UML to the software testing process. Many of the steps require research and effort to show their feasibility, but I believe that the effort is sorely needed.

4. The UML and Software Testing

There are many phases in the testing process, including unit, function, system, regression, and solution testing. The following table illustrates the differences between these phases, as well as the potential UML diagram for use in the phase.

Test Type	Coverage Criteria	Fault Model	UML Diagram
Unit	Code	correctness, error handling pre / post conditions, invariants	class and state diagrams
Function	Functional	functional and API behavior, integration issues	interaction and class diagrams
System	Operational Scenarios	workload, contention, synchron., recovery	use case, activity, and interaction diagrams
Regression	Functional	Unexpected behavior from new / changed function	SAME AS FUNCTION
Solution	Inter-System commun.	Interop. Problems	use case and deployment diagrams

However, the issue of using the UML in the testing process is not settled by merely noting which diagrams might be useful in various phases. Several issues must be resolved before the UML can be applied effectively across the testing process. I discuss two of these issues below.

The first issue concerns logistic issues in using UML models in the testing process. At first it is tempting to think that the models derived during design and implementation can be passed off for the testers to use "as is." This is not feasible for two reasons. First, the tester gains valuable insight by building or significantly modifying the models to be used in the testing process. Second, the models from the development process typically lack details and features that are required to develop test cases.

Given the need for specialized features for testing, the serious research task at hand is to determine the course of action for using the UML for testing. The conservative approach would be to define modeling libraries to standardize components for resuse in developing test models. This is a preferred solution when it provides enough expressive power for building models [4]. If more power is needed, extending the 4-layered metamodel would need to be investigated.

5. Conclusion

I presented an overview of both the UML and the use of OO artifacts in software testing. The current OO model based testing practices fall short of addressing the entire testing lifecycle, and I presented a framework for using the various UML diagrams in the testing process. Finally, I discussed the need for testers to create their own models, or to extensively modify the models from development in order to provide the features required. These features require special support, such as the development of libraries for building test models, or the modification of the UML metamodel to support testing activities.

6. References

1. Chillarege, R., Bhandari, I.S., Chaar, J.K., Halliday, M.J., Moebus, D.S., Ray, B.K., Wong M.Y. Orthogonal Defect Classification - A concept for in-process measurements. *IEEE TSE* 18, 11 (Nov. 1992).
2. Jacobson, Booch and Rumbaugh. *The Unified Software Development Process.* Addison-Wesley, Reading, MA, 1999.
3. Jorgensen, P. and Erickson, C. Object-oriented integration testing. *CACM* 37, 9 (Sept. 1994), 30-38.
4. Kobryn, C. UML 2001: A standardization odyssey. *CACM* 42, 10 (Oct. 1999), 29-37.
5. Perry, D. and Kaiser, G. Adequate testing and object-oriented programming. *JOOP* (Jan./Feb. 1990), 13-19.
6. Poston, R. Automated test from object models. *CACM* 37,9 (Sept. 1994), 48-58.
7. Rumbaugh, Jacobson, and Booch. *The Unified Modeling Language Reference Manual.* Addison-Wesley, Reading, MA, 1999.
8. Smith, M. and Robson, D. A framework for testing object-oriented programs. *JOOP* (June 1992), 45-53.