A Systematic Approach to Generate Test Cases based on Combinations of Information

Marisa A. Sánchez[1] and Miguel A. Felder[2]

[1] Universidad Nacional del Sur, Bahía Blanca
E-mail: mas@cs.uns.edu.ar
[2] Pragma Consultores, Buenos Aires
Url: http://www.pragma.com.ar
E-mail: mfelder@pragma.com.ar

1 Introduction

Software based systems incrementally provide critical services to users. Mobile telephone systems, for example, are used in circumstances in which the malfunctioning may have disastrous consequences. During the last years, software has been incorporated in devices used in daily life, such as audio and television. The diversity of systems in which software is incorporated is increasing. Thus, the software development process has to consider a variety of specification techniques and models, incorporating also techniques from engineering sciences. In particular, the validation and verification processes have to be adapted to these new developments. For example, the testing based solely on the software specification is incomplete. First, there is an implicit objective to verify that the program works correctly (as in the testing model of the 1957–1978, [GH88]). Myers [Mye83] says that with the aim of demonstrating that a program does not fail, we can unconsciously select data that has a low probability of exposing faults. On the other hand, if the objective is to demonstrate that a program has faults, test data will have a higher probability of revealing them. In specification-based testing we select data for which the desired behavior for the system is defined. As stated by Boris Beizer [Bei95] testing should include both clean and dirty tests. Dirty tests are designed to "break" the software; clean tests are designed to demonstrate that software executes correctly. Specifications only provide clean tests.

It is clear that a system has more possible behaviors than those deduced from the specification. The problem is that, given the diversity of information that we have to consider to understand a system, it is not obvious how to define those behaviors. In this work, we propose a systematic approach generate test cases based on combinations of information:

- information based on the software specification;
- information about the behavior of other system components, such as, memory resources, network availability, deadlock of resources;
- information about different operative conditions.

2 Research direction

The main problems that we encounter when we try to deduce information outside the specification are the following:

(a) For the case of specification-based testing, the number of possible behaviors is bounded by what is described in the specification. If we also consider information outside the specification, the number of possible behaviors is infinite.
(b) We have to deal with specifications provided in different languages, with different levels of granularity and abstraction, and that they consider different views of the system.

In Sections 2.1 and 2.2 we discuss these points.

2.1 Selection of "dirty" behaviors

To address the first point, we propose to characterize possible behaviors and to give a priority according to some criteria. We use Fault Tree Analysis to determine how an undesirable state (failure state) can occur in the system [us881]. This analysis is a widely used technique in industrial developments, and allows to describe how individual component failures or subsystems can combine to effect the system behavior.

A fault tree consists of the undesired top state linked to more basic events by logic gates. Once the tree is constructed, it can be written as a Boolean expression and simplified to show the specific

combinations of identified basic events sufficient to cause the undesired top state. The sets of basic events that will cause the root event are regarded as Minimal Cut Sets.

2.2 Integration of different sources of information

Concerning the second point mentioned in the introduction of this section, we have to integrate Fault Tree Analysis results with statecharts. We assume we have a specification of the desired behavior for the system using statecharts [Har87]. This formalism is widely used within the software engineering community, and has been adopted by the Unified Modeling Notation (UML) [DH89,HLN+90,BRJ98].

The results of the Fault Tree Analysis are related to the system specified behavior to determine how we can reproduce these scenarios. In our work we propose to interpret each Minimal Cut Set as a Duration Calculus formula [ZHR91]. We defined some conversion rules of a formula to a statechart. These rules are applied to the syntactic categories of Duration Calculus formulas. By combining both sources of information within a common semantic framework, we can systematically build a testing model. The testing model provides a representation of the way the system behavior can be compromised by failures or abnormal conditions or interactions.

In particular, if this conditions refer to a peak activity that exceed system limitations, we are doing stress testing. Stress testing evaluates the behavior of systems that are pushed beyond their specified operational limits [Ngu01]. Stress testing requires an extensive planning effort for the definition of workload, and this involves the analysis of different components that effect system behavior (*e.g.* memory resources, network bandwidth, software failures, database deadlocks, operational profiles). However, this analysis is usually performed ad hoc. We propose to use Fault Tree Analysis helps to define workload scenarios. The results of this analysis are composed with the specification statecharts, and we obtain a model that describes how a given workload can be reproduced.

2.3 Reduction of the testing model

The testing model is specified using statecharts. Although its semantics is very intuitive, the inherent complexity of many of today's

applications may lead to large and complex statecharts. To address this problem, we propose to reduce statecharts using slicing techniques.

Program slicing is a technique for decomposing programs by analyzing their data flow and control flow. The traditional definition of slicing is concerned with slicing programs written in imperative programming languages [Wei84]. Therefore, it is assumed that programs contain variables and statements, and slices consist solely of statements. Sloane *et al.* extended the concept of slicing to a generalized marking of a program's abstract tree [SH96]. This generalization allows slicing based on criteria other than the use of a variable at a given statement. We also base our approach to slicing on a marking of the abstract syntax tree, and for this purpose we define a formal grammar to describe correct syntax for statecharts. A slicing criterion of a statechart is defined by a state. The criterion determines a projection on the sequences of the statechart that throws out all states and transitions that do not contribute to reach the state of interest.

2.4 Tool support

The testing process can become very tedious, if not unpractical because the amount of information needed to describe a test case is large in most real problems. So it becomes necessary to support any testing approach with tools, otherwise it may be useless for the practitioner. We direct our efforts towards developing an approach that requires as little human intervention as possible. Thus, the resulting approach allows a level of automation that can significantly enhance the productivity of the testing process. Much of the ongoing work is directed at developing tool support.

References

[Bei95] Boris Beizer. *Black-Box Testing. Techniques for Functional Testing of Software and Systems.* John Wiley Sons, Inc., 1995.

[BRJ98] G. Booch, J. Rumbauch, and I. Jacobson. *The Unified Modeling Language. User Guide.* Object Technology Series. Addison Wesley Longman, Reading, MA, USA, 1998.

[DH89] D. Drusinsky and D. Harel. Using Statecharts for Hardware description and Synthesis. *IEEE Transactions on Computer-Aided Design*, 8:798–807, 1989.

[GH88] David Gelperin and Bill Hetzel. The Growth of Software Testing. *Communications of the ACM*, 31(6):687–695, June 1988.

[Har87] D. Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Prgramming*, 8:231–274, 1987.

[HLN+90] D. Harel, H. Lachover, A. Naamad, A. Pnueli, M. Politi, R. Sherman, A. Shtulltrauring, and M. Trakhtenbrot. Statemate: A Working Environment for the Development of Complex Reactive Systems. *IEEE Transactions on Software Engineering*, 16, 1990.

[Mye83] G. Myers. *El Arte de Probar el Software*. Librería El Ateneo Editorial, Buenos Aires, 1983. Spanish translation.

[Ngu01] Hung Q. Nguyen. *Testing Applications on the Web*. John Wiley and Sons, Inc., 2001.

[SH96] A. M. Sloane and J. Holdsworth. Beyond traditional program slicing. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 180–186, 1996.

[us881] Fault Tree Handbook. Nureg-0492, U.S. Nuclear Regulatory Commission, Washington, D.C., Jan. 1981.

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, 10(4), July 1984.

[ZHR91] Chaochen Zhou, C.A.R. Hoare, and Anders P. Ravn. A Calculus of Durations. *Information Proc. Letters*, 40(5):269–276, Dec. 1991.