A&QT-R 2002 (THETA 13)
2002 IEEE-TTTC International Conference on Automation, Quality and Testing, Robotics
May 23-25, 2002, Cluj-Napoca, Romania

GENERATING UML TEST CASES FOR SYSTEM-LEVEL TEST

Andrea BALDINI, Alfredo BENSO, Paolo PRINETTO

Politecnico di Torino
Dipartimento di Automatica e Informatica
Corso Duca degli Abruzzi 24, I-10129, Torino, Italy
Email: {baldini, benso, prinetto} @polito.it

Abstract

The goal of this research is the study of a methodology to convert design level specifications to system level functional test cases of complex systems, using de jure and de facto industrial standards (Unified Modeling Language, UML) and advanced techniques from academic research.

The research focuses on a formal framework for the definition of the process, exploring for the first time the first part of the process, the actual generation of the test sequences from a UML model.

The process heavily relies on messages in order to increase both abstraction and descriptive power; the target is the generation of design-level messages that can be then translated into test sequences.

Experimental results are also presented to discuss the feasibility of the study.

Keywords

Unified Modeling Language (UML)
Requirements and Specifications Modeling
System Properties
System Level Testing

1 INTRODUCTION

The aim of our research is the generation of system level test sequences for real-time embedded systems, starting, or better re-using, the system specifications written at the beginning of the design process.

Our starting point is a basic definition: “system testing is the process of testing an integrated hardware and software system to verify that the system meets its specified requirements” from IEEE Standard Glossary of Software Engineering Terminology, IEEE/STD610.12-1990, Institute of Electrical and Electronic Engineering, 1990.

The first step is so a definition of requirements and specifications in some way that is in the middle between a pure formal representation and a too weak non-formal and textual description of the system. What we need is a standard notation, and we have already observed [1][2] that Unified Modeling Language (UML) [3] provides a quite powerful one. UML allows the designer to express the specification in a standard graphical notation, providing different views of the system at different levels of abstraction and detail.

The UML model of the system becomes the input of the entire process.

The process is implemented into a test engine [2]. The test engine must be able to read the input UML diagrams and generate the final test sequences. The final target is on one side an embedded system under test, and on the other side a set of commonly available simulators and actuators to run the test sequence. The test environment is so the combination of the test engine and all simulators and actuators driven by the test engine.

In this paper we will not address the translation phase, but we will concentrate on the generation phase.

Our attention focuses on embedded systems with medium to high complexity, and in other researches in collaboration with Magneti Marelli [1][2] we have gained a considerable expertise in real industrial products testing.

2 STATE-OF-THE-ART

UML is a standard of the Object Management Group (OMG) from 1997. Initially it was thought to describe software systems but the idea of modeling not only pieces of software, but entire and complex embedded systems in UML is quite recent but not new in literature [3][4][5][6]. Industrial tools (Rational / I-Logix) are also available to support the standard, and specific tools have been developed in academics, in particular for hardware support [7].

Since UML is only a notation, and does not include a process, it is not a complete methodology. General (RUP [2]) and specific (ROPES [3], BOOM [8]) methodologies have been so developed both for software and embedded real-time systems, and at least one (BOOM [8]) is specifically dedicated to behavioral description of the system.

These processes address partially test issues, but none has a real design-to-test approach; moreover, some other solutions are specifically addressed to different models, such as SDL (Autolink [9]) and pure statecharts models [10][11]. These last researches are complementary to our approach, even if they tend to lose the generality of notation of UML and are test-case oriented.

In this research we have used some results from other researches on UML and test generation [12][13], we will cite them in the following sections.

UML-and-test topics are so an open problem, and the market itself seems to feel the necessity of more refined models and methodologies both at theoretical and practical levels, in particular aiming at improving functional end-of-line tests.

3 PROCESS DEFINITION

The gap between the specification level and the test level (test simulators) is terribly wide, so we have decided to implement the process using two consecutive steps.

The overall process uses as input a well-formed UML model of the system, including all the information about the functional requirements of the system, i.e. the behavior. The result is a timed collection of messages to feed into a test environment.

The first step of the process is the generation of a test set T expressed at high level of abstraction, whereas the second step of the process translates such high level description into a lower level description directly understandable by the test environment. The result of the first step is so the input of the second one, we call it design-level test set (see Figure 1).

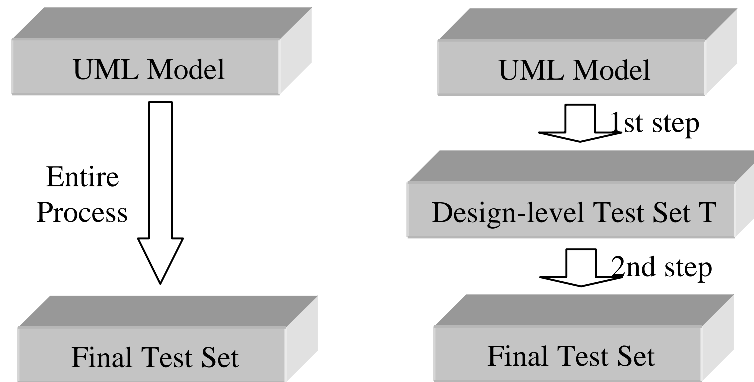

Figure 1: division of the process

From a general perspective, the design-level test set T is a set of test cases; we define a design-level test case t as an ordered list of messages, with optional timing constraints.

The obvious way to represent such a list is a UML sequence diagram. A sequence diagram is a graphical representation of a sequence of messages exchanged between objects. A message is a specification of a stimulus, in other words, communication between a sender and a receiver. The message specifies the roles played by the sender object and the receiver object and it states which operation should be applied to the receiver by the sender (see Figure 2).

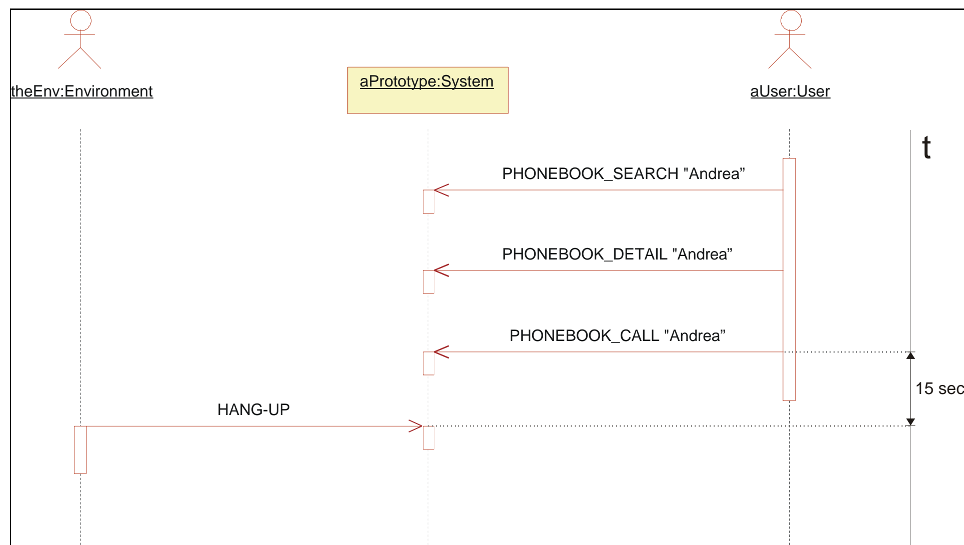

Figure 2: example of Design-level test sequence as sequence diagram

4 TEST GENERATION

The second part of the process (translation phase) has already been addressed in other papers [1][2] and has been deeply analyzed. However in those papers we always supposed to have directly as input such design-level sequence diagrams.

Now we are interested in extending the process toward the generation phase, to indicate techniques directly usable by the test engineer to take a UML model and generate the design-level test set T. We will analyze some criteria we have used to

generate the test sequences and that seems to be quite effective. Partial automatism of such techniques is possible, even if we manually generated the test cases in this case.

In the following sections we will consider the main diagrams of UML models and incrementally generate test set T.

4.1 USE-CASE DIAGRAMS

The first diagrams of UML models are use case diagrams: the system is described from the user perspective and at very high level of abstraction. Each use case is expressed as a set of scenarios, each representing a possible use of the system.

Scenarios are classified in default scenario(s), the usual operative condition of the system, possible alternatives occurring in particular cases and erroneous scenarios catching the common the most common mistakes in the interaction between the external world and the system.

In this case the criterion to generate the test set T is the following:

For each use-case diagram in the model, T must include test cases that execute each possible scenario of each use case in the diagram.

Such generation is not always trivial, because in the case of erroneous scenarios the test case t must effectively create the external conditions able to simulate the error.

Generally speaking it is not possible to guarantee that such test cases provide a good coverage; the test cases will just be those that the designers considered, which are the ones the system is most likely to execute correctly.

More accurate results can be obtained considering more precise views of the system, such as the interaction diagrams, i.e. sequence and collaboration diagrams.

4.2 INTERACTION DIAGRAMS

In UML models, objects interact to implement behavior. This interaction can be described in two complementary ways, one focused on individual objects and the other on a collection of cooperating objects. The collective behavior of a set of objects can be modeled in terms of how they collaborate. A collaboration is a description of a collection of objects that interact to implement some behavior within a context.

An interaction is a behavioral specification that is composed of a sequence of communications among a set of objects within a collaboration to accomplish a specific purpose. Each interaction contains an ordered set of messages. A collaboration diagram is a graphical representation of a collaboration, including the interactions among objects. Each collaboration diagram represents a complete trace of messages during the execution of an operation, that is, a message sequence path.

We define the following criterion to generate the test set T, derived from similar criterion for object-oriented software [12]:

For each collaboration diagram in the model, T must include at least one test case t that executes the message sequence path of the collaboration diagram.

Sequence diagrams are almost equivalent to collaboration diagrams, but they can more easily describe branches and choices inside an operation. In other words, they can contain more than one message sequence path. We must define so an additional criterion to generate the test set T, derived from the previous one:

For each sequence diagram in the model, T must include test cases that execute each possible message sequence path of the sequence diagram.

We must emphasize that considered sequence diagrams are those inside the model, and they are quite different from the design-level test sequences inside the test

set T. Such test cases are expressed in form of sequence diagram, but uses different messages and a lower abstraction level.

4.3 STATECHARTS

UML statecharts are based on finite state machines using an extended Harel state chart notation, and are used to represent the behavior of an object. As they are the most formalizable aspects of UML, statecharts provide a natural basis for test data generation.

The state of an object is the combination of all attribute values and objects the object contains. The dynamics of objects are modeled through transitions among states. The UML syntax for transitions is:

event-name (parameters) [guard] / action list ^ event list

Event-name is a label for the transition with its associated parameters; guard is a precondition that controls whether the transition is taken or not, and action list and event list define changes that occur as a result of the transition.

Four kinds of events can be specified in UML: call events, signal events, time events, and change events. A call event represents the reception of a request to synchronously invoke a specific operation. A signal event represents the reception of a particular (synchronous) signal. A time event represents the passage of a designated period of time after a designated event. A change event models an event that occurs when an explicit boolean expression becomes true due to a change of one attributes.

Since we want to generate test data for all the four kinds of events, we use three separate criteria that are used on each statechart of the model to generate the test set T:

Call-event and signal-event transitions: for each transition, T must include test cases that cause the event to be triggered (call or signal); if a guard condition is present, both true and false guard preconditions must be tested;

Time-event transitions: for each transition, T must include test cases that indicate an explicit time condition able to trigger the time event;

Change-event transitions: for each Boolean predicate P on each transition, T must include test cases that cause each clause c in P to result in a pair of outcomes where the value of P is directly correlated with the value of c.

The first criterion states that normal transitions should be triggered by at least a test case, and if a guard is present, at least one test case must result in a true guard condition and at least one in a false guard condition (transition not triggered). The second criterion simply covers all the time-event transitions. The third criterion uses full predicate coverage [13] for change event transitions; full predicate coverage is based on the philosophy that each clause should be tested independently, that is, while not being influenced by the other clauses. In other words, each clause in each predicate on every transition must independently affect the value of the predicate.

5 EXPERIMENTAL RESULTS

We have used all the presented criteria to generate test cases for a complex real-world application, a top-of-the-line car console used in modern vehicles to integrate functionalities such as phone, radio, CD, messaging, positioning and navigation, alarms and common car data. We have obtained quite a high number of test cases:

Diagrams	Use case	Interaction	Statecharts	Total
Number	79	117	356	485

It is evident that the more informative is the diagram, the higher is the number of test cases that can be obtained from such diagram. So it is natural that the test set related to statecharts is the biggest and the most significant, but also expensive in terms of time.

Note also that the total cardinality of the test set is lower than the sum of all the test cases, because some test cases are actually equal in more than one set.

Each criterion has been fully exploited and fully covered.

6 CONCLUSION

The process we have analyzed and described is able to generate test cases in terms of sequences of design level messages that can then be translated into a list of timed test commands for a specific test environment. The system is a black-box and the description language is UML, representing a good compromise between simplicity and formality. The soundness and the feasibility of the process are guaranteed by a formal framework and by experimental verification. Main advantages are the use of a standard notation accepted in the industrial world and the formality of the criteria. The limitations are mainly related to the quality of the model, so that it is difficult to precisely evaluate the final coverage of the test set.

7 REFERENCES

[1] Baldini, A.; Benso, A.; Mo, S.; Taddei, A.; Prinetto, P. [2001] – Towards a Unified Test Process: from UML to End-of-Line Functional Test – *IEEE International Test Conference 2001 (ITC'01) - Proceedings* - page(s) 600-608 - Oct. 2001

[2] Baldini, A.; Benso, A.; Mo, S.; Taddei, A.; Prinetto, P. [2002] – Beyond UML to an End-of-Line Functional Test Engine – *Design, Automation and Test in Europe (DATE'02) – Proceedings* – March 2002

[3] Douglass, B.P. [1999] – Real Time UML – Addison Wesley Pub. - 1999

[4] Selic., B. [1999] – Using UML for Modeling Complex Real-Time Systems – Rational Inc. White Paper – *available online at www.rational.com*

[5] Fernandes, J.M.; Machado, R.J.; Santos, H.D. [2000] – Modeling industrial embedded systems with UML – *Hardware/Software Codesign, 2000. CODES 2000. Proceedings of the Eighth International Workshop on* – page(s): 18 – 22 - May 2000

[6] Selic, B. [1998] – Using the object paradigm for distributed real-time systems – *Object-Oriented Real-time Distributed Computing, 1998. (ISORC 98) Proceedings. 1998 First IEEE International Symposium on* – page(s): 478 – 480 - April 1998

[7] Sinha, V.; Doucet, F.; Siska, C.; Gupta, R.; Liao, S.; Ghosh, A. [2000] – YAML: a tool for hardware design visualization and capture – *System Synthesis, 2000. Proceedings. The 13th International Symposium on* – page(s): 9 – 14 - Sept. 2000

[8] Mendelbaum, B.H.G.; Gallant, R.; Brette, J.-F.; Ducateau, Ch.F. [2000] – Java-prototyping of hardware/software CBS using a behavioral OO model – *Eng. of Computer Based Systems, 2000. (ECBS 2000) Procs. 7th Intl. Conference and Workshop on the* – page(s): 73 – 81 - Apr. 2000

[9] Koch, B.; Grabowski, J.; Hogrefe, D.; Schmitt, M. [1998] – Autolink-a tool for automatic test generation from SDL specifications – *Industrial Strength Formal Specification Techniques, 1998. Proceedings. 2nd IEEE Workshop on* – page(s): 114 – 125 - Oct. 1998

[10] Kim, Y.G.; Hong, H.S.; Bae, D.H.; Cha, S.D. [1999] – Test cases generation from UML state diagrams – *Software, IEE Procs* – page(s): 187–192 Aug. 1999

[11] Li Liuying; Qi Zhichang [1999] – Test selection from UML Statecharts – *Technology of Object-Oriented Languages and Systems, 1999. TOOLS 31. Proceedings* – page(s): 273 – 279 - Sept. 1999

[12] Aynur Abdurazik and Jeff Offutt [2000] - Using UML Collaboration Diagrams for Static Checking and Test Generation - *The Third International Conference on the Unified Modeling Language (UML'00)*, pages 383-395, York, UK, October 2000.

[13] Jeff Offutt and Aynur Abdurazik. [1999] - Generating tests from UML specifications - *Proceedings of the Second IEEE International Conference on the Unified Model-ing Language (UML99)* - pages 416-429 - Oct 1999