UML-based Test Generation and Execution

Jean Hartmann, Marlon Vieira, Herb Foster, Axel Ruder
Siemens Corporate Research, Inc.
755 College Road East
Princeton NJ 08540, USA
jeanhartmann@siemens.com

ABSTRACT

The Unified Modeling Language (UML) is gaining wide acceptance in industry and we, at Siemens Corporate Research, are seeking ways in which UML diagrams can be used as the basis for functional testing. Our approach focuses on developing effective techniques and tools for test case generation and coupling them with suitable execution tools for unit, integration and system testing. For example, our current approaches focus on integrating our techniques with the Rational Rose UML modeling tool and unit test execution tools such as Junit and system test tools, that is, capture-replay tools such as Compuware Testpartner. The goal of our approach is to generate a set of black-box conformance tests that can be used to validate a component, subsystem or application under test.

1. Introduction

Improvements in software testing techniques are needed to address the increasingly complex applications that are implemented by today's software systems. The UML and its diagrams are widely used to visually depict the static structure and more importantly for us, the dynamic behavior of such applications. This trend provides us with an excellent opportunity to meld our proven test generation technology (TDE) with the UML and give developers and testers the capability of automatically generating black-box conformance tests early on.

For unit and integration level testing, we can derive tests from UML Statecharts and Sequence diagrams, which embody the behavioral description of each component of interest and more importantly, the interactions or events between them. At the system level, we focus on UML Use Cases and Activity diagrams to convey the overall application behavior with users.

In both cases, the diagrams are annotated with textual references, which represent the test requirements, that is, the data variations or choices as defined by the category-partition method. From these annotated diagrams, the test cases can then be derived using our test generation engine and executed with the help of the appropriate unit/integration or system test tool. Our approach aims at minimizing the testing costs, time and effort associated with initially developing customized test drivers, test stubs, and test cases as well as repeatedly adapting and rerunning them for regression testing purposes at each level of system integration. In this paper, we give an overview of some important aspects of our UML based testing techniques. However, for this workshop, we will focus on the approach for automatically generating and executing system tests.

In Section 2, we present an overview of the category-partition method, which is the basis for our UML test generation methodology. Section 3 discusses our approach for unit and integration level testing using UML statechart diagrams. In Section 4, describes our approach for system level test generation. Section 5 presents conclusions and future work.

2. Category-Partition Method

Underlying all of our UML-based testing approaches is the category-partition method that was developed at Siemens Corporate Research [1,2]. The method is embodied by the Test Specification Language (TSL) and realized by the Test Development Environment (TDE) tool. Thus, TSL is the notation used for writing test designs and TDE is the tool used for generating test scripts, executable or otherwise, from the test designs.

The category-partition method identifies behavioral equivalence classes within the structure of a system under test. A category or partition is defined by specifying all possible data choices that it can represent. Such choices can be either data values or references to other categories or partitions, or a combination of both. The data values may be string literals representing fragments of test scripts, code, or case definitions, which later can form the contents of a test case. In TSL, the type of a data value is dynamically determined by the context in which it is used. There are essentially five basic types: identifiers, numbers (integer and floating point), Booleans, strings, and lists. If TSL does not support a complex data structure directly, that is, the desired structure is not defined in the TSL's pre-defined functions or implemented data structures, then we can use the added power of TSL's embedded language (Tcl) to create **{**xe "Action block"**}**action blocks that can allow us to describe the desired structure. Tcl has a rich set of pre-defined functions and operators. It also supports a fairly complete set of commands for building and manipulating list structures.

However, when applying the UML-based approaches below much of the knowledge required in creating TSL test designs is hidden from users, making these approaches much more user-friendly and widely applicable.

3. Unit and Integration Testing

We have developed an approach for unit and integration testing, which focuses on testing the APIs of software components that form part of the server-side application or business logic [3,4]. It is important to ensure that components are delivered in conformance with their specifications, which typically consist of definitions of their interfaces and the legal order in which operations may be invoked on them. It is also important to validate if the individual components are correctly integrated into the system. While each component may have behaved correctly during unit testing, it may not do so when interacting with other components due to, for example, interface mismatches.

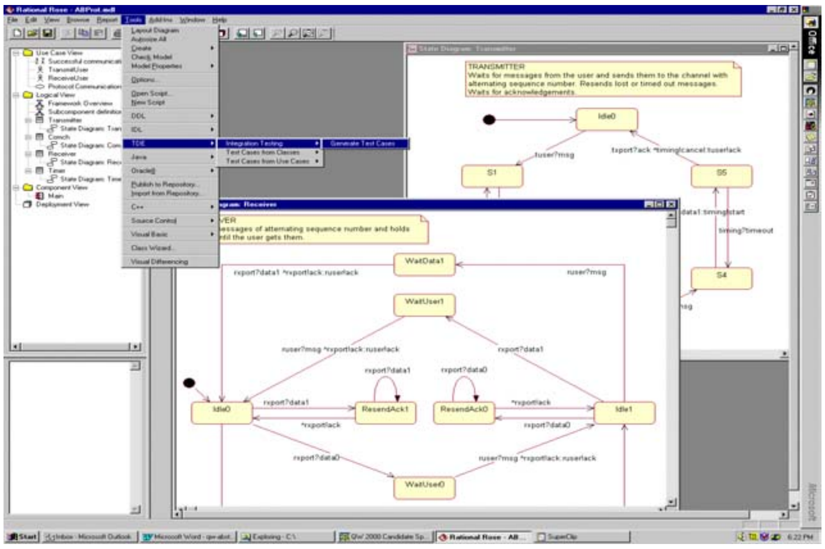
Currently, component interfaces and their protocol specifications are being described or modeled in a variety of ways. For example, in the case of the Enterprise Java Beans Specification, this is achieved through contracts and Sequence Diagrams. While a UML Sequence Diagram is useful at describing a specific component interaction scenario, it may require a large number of such diagrams to completely specify the interaction of a complex component with its client(s). A more concise and compact way is to represent such scenarios is to depict them via a UML Statechart Diagram. Our approach requires the formulation of a UML Statechart Diagram for each component and the modeling of events or interactions between components using a CSP-like notation and the composition of these Statecharts. Such a composed Statechart model represents a global behavioral model for a set of integrated components and is automatically created using incremental composition techniques. In this global behavioral model, the significant properties, that is, behaviors, of the individual state machines are preserved. Particular consideration was given to addressing the issues of scalability and complexity with respect to the composition method. Using this model, tests cases are generated for unit and integration testing.

Before our TDE/UML test generator can create test cases, the UML Statechart Diagram has to be annotated by the user with additional test data such as coverage requirements, constraints and preconditions. Once this information has been added to the diagram, our tool is able to generate the corresponding TSL test design, and then compile this test design to create the test cases. A TSL test design is created from a statechart or a global behavioral model (composed statecharts) by mapping states and transitions to TSL categories or partitions, and choices. States are the equivalence classes and are therefore represented by partitions. Each transition from the state is represented as a choice of the category/partition. Only partitions are used for equivalence class definitions, because paths through the state machine are not limited to certain outgoing transitions for a state; this would be the case when using a category. Each transition defines a choice for the current state, combining a test data string (the send and receive event) and a reference to the next state. A final state defines a choice with an empty test data string.

TDE creates test cases in order to satisfy all specified coverage requirements. Input sequences for the subsystem are equivalent to paths within the global behavioral model that represents the subsystem, starting with the initial states. Receive transitions with events from external connections stimulate the subsystem. Send transitions with events to external connections define the resulting output that can be observed by the test execution tool. All communication is performed through events.

While TDE/UML is capable of generating a set of executable test cases, it does not execute them. For this purpose, we rely on generic unit testing tools such as Junit for Java, CppUnit for C++ or our own specialized C++ and COM-specific unit test harness, TECS [5]. The TECS tool includes the following features: (1) Test Harness Library – this is a framework that provides the basic infrastructure for creating the executable test drivers; (2) Test Case Compiler – it is used to generate test cases from a test case definition ; (3) Sink Generator – it is used to generate C++ sink classes out of an COM IDL interface definition file, and (4) Test Control Center – it provides the developer a way of running test cases interactively through a graphical user interface or in batch mode. The information generated during test execution is written into an XML-based tracefile. The Test Control Center provides different views of this data such as a trace view, an error list, and an execution summary.

Together, these two tools, TDE/UML and TECS, form the UML-based test environment known as T*n*T.


TRANSMITTER
Waits for messages from the user and sends them to the channel with alternating sequence number. Resends lost or timed out messages. Waits for acknowledgements.

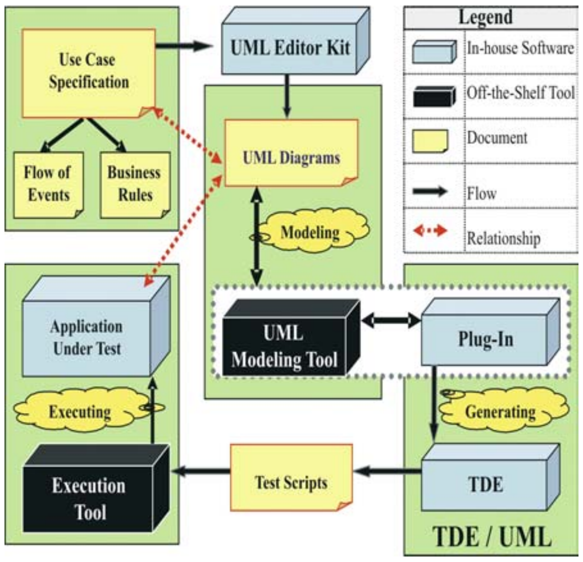
VER
essages of alternating sequence number and holds
ntil the user gets them.

3. System Testing

We have also developed an approach for automatically generating and executing system tests for command-line or User Interface-based applications [6]. With this approach, tests are automatically generated from behavioral models of the application, that is, from UML Use Cases and Activity Diagrams and then executed using a UI capture/replay tool. This approach encompasses the following tasks:

1. Modeling system behavior, that is, describing it from scratch or semi-automatically converting existing textual use case specifications into the appropriate UML models.
2. Generating an executable set of test cases or test scripts using the above explicit visual models of system behavior.
3. Executing these test scripts and performing verification, both manually and automated.

The figure below shows how the different tasks described above have been realized as a suite of tools. Initially, users can either model directly in Rational Rose or make use of our UML Editor Kit to semi-automatically convert an existing set of use case specifications written in Microsoft Word into models readable by Rational Rose, the UML visual modeling tool. Users then annotate and refine the UML diagrams in preparation for test generation. Our TDE test generator is then executed in order to generate the required set of test cases in XML (EXtensible Markup Language). These test cases are then in turn converted to a set of executable test scripts via an appropriate XSL (Extensible Stylesheet Language) template and executed using the UI test execution tool of choice.



Our approach considers testing interactive systems as the process of verifying sequence of user actions and system responses, that is, the relationships between a specific user action and the subsequent system response. Although the use case diagram gives an idea as to the type of functionalities that the system performs, it doesn't represent the order in which these use cases might occur or how the interactions between actors (i.e., users) and the system are conducted. Activity diagrams are generally suitable to model these sequences and interactions, since the diagrams represent the flow driven by external events and internal processing.

In order to be useful for test generation, an Activity Diagram must include annotations in the form of UML stereotypes (and notes) and guard conditions specification; those annotations must adhere to certain format rules. Each activity, shown by an elliptical box in the UML activity diagram, must be annotated with one of the following stereotypes: <<UserAction>>, <<SystemResponse>> or <<Include>> to indicate whether it is a user or system activity, while the latter stereotype enables the test generator to replace this activity with the entire activity diagram for the use case specified. Depending on the complexity of the system under test, it may be necessary to specify expressions in the guard conditions, which govern conditional flows. In this case, our approach requires the test designer to define variables in the diagram using the <<define>> stereotype. This variable definition list is represented as a text label in the diagram and must be attached to the <<UserAction>> activity where the variable has its origin.

A TSL test design is created from the activity diagram by mapping its activities and transitions to TSL partitions and choices. It is important to realize that the control flow within the diagram is totally determined by the diagram variables. These represent the various user inputs and the part of the system state that is relevant for this particular use case. Thus, every test case is defined by choosing values for all the variables in the diagram. With respect to the category-partition method, every variable has to be mapped to a partition, which divides the value range of the variable according to its equivalence classes. Besides variable partitions, a partition is created for every activity and a choice within the partition for every outgoing transition.

We have also developed support for the test execution process that focuses on capturing, modularizing and parameterizing a representative set of executable test scripts that can be run against the system under test using a commercial UI testing tool.

The original T*n*T environment was thus enhanced to support this additional approach. T*n*T is now capable of not only generating test cases from Statechart and Sequence Diagrams, but also Use Cases, that is, Activity Diagrams. The environment has been updated to work with Rational Rose 2003 and Compuware TestPartner 3.0, although other UI testing tools could be used.

5. Conclusions and Future Work

In this paper, we have presented an overview of an ongoing research and development project at Siemens Corporate Research in which UML-based models are being used to improve the testing of components, subsystems and applications. We have summarized different aspects of our approach and discussed their implementation.

Our approach benefits from the use of COTS tools, such as Rational Rose and Compuware TestPartner, and the mature, in–house tools such as TDE. Together, they provide a solid basis to continue conducting our anticipated empirical study on the effectiveness and efficiency of this approach.

Several topics for future research have been identified. Among them are ways to: (1) improve the performance of the test case generation step and the reliability of the scripts executed, and (2) implementation of a more precise measurement technique for data coverage, particularly when we want to improve the process of test data creation and utilization of the data during test execution.

Supporting test generation based on the upcoming UML2.0 standard is also an important research activity. We are adapting the approaches described above, for example, to the UML 2.0 Test Profile. Two possible approaches are being prototyped. These include: (1) using TDE/UML to generate test context, test behavior and data for specific test objectives in the UML Testing Profile; (2) annotating the UML Testing Profile test behavior description (State and Activity Diagrams) to optimize test case generation.

Acknowledgements

We would like to thank Dr. Jürgen Kazmeier, the Head of the Software Engineering Department at Siemens Corporate Research for his continued support of our work as well as Professor Manfred Broy and Michael Meisinger at the Technical University, Munich, for their cooperation.

6. References

[1] M. Balcer, W. Hasling, and T. Ostrand, “Automatic Generation of Test Scripts from Formal Test Specifications”, Proceedings of ACM SIGSOFT'89 - Third Symposium on Software Testing, Verification, and Analysis (TAVS-3), ACM Press, pp. 257-71, June 1990.
[2] T.J. Ostrand and M. Balcer, ‘The Category-Partition Method For Specifying and Generating Functional Tests”, CACM, Vol. 31, No.6, June 1988.
[3] J. Hartmann, C. Imoberdorf, and M. Meisinger, “UML-Based Integration Testing. Proceedings of ISSTA 2000, Aug. 2000, pp. 60-70.
[4] M. Meisinger, Automatic Test Case Generation From Communicating State Machines, M.S. Thesis, Technical University Munich, 2000.
[5] J. Hartmann, “Testing and Debugging of COM Components”, Windows Developer Network, June 2002.
[6] J. Hartmann, M. Vieira, H. Foster and A. Ruder, “TDE/UML - A UML-based Test Generator to Support System Testing”, submitted to Intl. Conference on Automated Software Engineering 2004.