Model-driven System Testing for Interactive Applications

Marlon Vieira and Johanne Leduc
Siemens Corporate Research
755 College Road East
Princeton, NJ 08540

{marlon.vieira, johanne.leduc}@siemens.com

Abstract

This paper describes an approach for automatically generating system tests from behavioral models of an application using the Unified Modeling Language (UML.) The proposed approach builds on and combines existing techniques for data coverage and graph coverage. It first consists of using the Category-Partition method to introduce data into the UML model. UML Use Cases and Activity diagrams are used to respectively describe which functionalities should be tested and how to test them. This combination has the potential to create a very large number of test paths.

This approach offers two ways to manage the number of tests. First, custom annotations and guards use the Category-Partition data which allows the designer tight control over possible, or impossible, paths. Second, automation allows different configurations for both the data and the graph coverage. The goal of this paper is to illustrate the benefits of our automated, model-based approach for improving system test design, generation and automation.

1 Introduction

With testing activities accounting for a large part of the total effort of a software life cycle, there can be no question of their expense. Though advanced development processes and tools have helped organizations reduce the time to build products, they have not yet been able to significantly reduce the time and effort required to test them. Clearly, there is a need for improvement in testing support. Development has raised the level of abstraction through the use of models: why not do the same for testing? Advancing this notion is the trend towards the increased use of model-based development (UML) which will allow for automatic test case generation. Few existing approaches are based on Activity diagrams; however these use only the control flow aspect. As far we know, our tool TDE/UML is the only approach (research project) that can generate test cases based on Activity Diagram control flow and on data coverage.

In our approach, we proceed from modeling, to adding testing information then choosing the configuration and finally generation and saving/managing tests (preparing for execution). Section 2 describes the models of the system behavior required by our approach. Our automation tool is briefly described in section 3 and an example is given in section 4. Throughout this paper, we use an example to demonstrate our approach: the "Letter Wizard" for Microsoft Word™, which helps users to write and modify correspondence using a graphical interface.

2 Modeling System Behavior

In this section, we describe the derivation of system behavior and its depiction as UML Diagrams. We make use of UML Use Case diagrams to describe the relationship among the diverse use cases specified for the system and the actors who interact with the system according to those use cases. UML Activity Diagrams are used to model the logic captured by a single use case. The set of activity diagrams represents the overall behavior specified for the system and is the basis for testing the different functionalities and business rules described in the use cases specification. The test generation will consist of determining all the paths through the activity diagrams.

The activity diagrams are then annotated with the categories as defined in the data coverage section (section 2.2). These annotations are in the form of notes anchored to a particular activity in the diagram and use custom stereotypes. The data inputs from these notes are combined with the generated test paths. This is how our approach achieves data and graph coverage.

2.1 Use Cases

UML Use Case Diagrams are used to represent the functionality of the system from a top-down perspective. Each use case provides one or more scenarios that convey how the system should interact with the end user or another system to achieve a specific business goal.

In our approach, each use case must have an associated activity diagram. If the use case has included or extended other use cases, these must be represented in the diagram as activities of the same name. Figure 1 shows the Use Case Diagram for our Microsoft Word Letter Wizard example. Though it is possible that this diagram is an excerpt of a larger model, it is irrelevant for our chosen example as we are only testing this use case. The top level use case includes four other use cases, each representing a tab of the user interface.

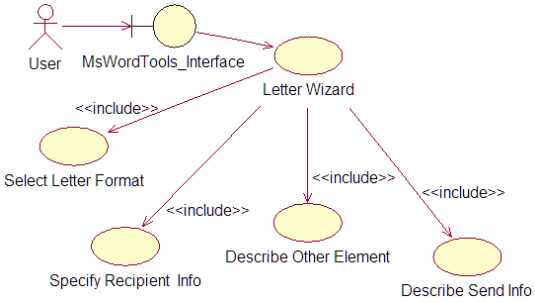

Figure 1: Example Use Case Diagram

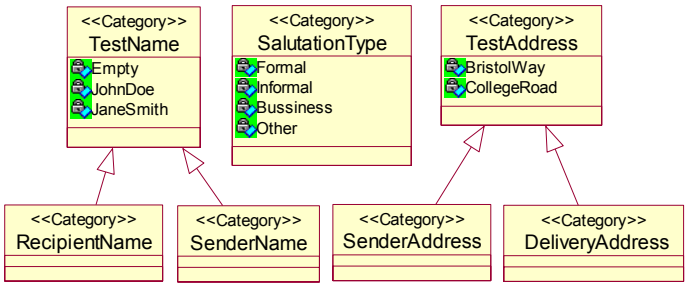

Figure 2: Example Class Diagram

2.2 Data Coverage

Underlying our UML-based testing approach is the category-partition method that was developed at Siemens Corporate Research [6]. 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, references to other categories or partitions, or a combination of both. The data values become inputs to test cases and can be elements in guard conditions on transitions. In our approach, categories and partitions are represented by classes with the stereotype <<Category>> and <<Partition>> respectively. The data choices for each category or partition are the attributes of the class.

Figure 2 shows an excerpt of the classes for our Letter Wizard example. Most of these categories are used in the "Specify Recipient Info" use case. This use case will require input data: a recipient name, a delivery address and a salutation type. For each of these categories, the possible data choices are presented as the attributes: three choices for the name, four for the salutation type and two for the address. These must be chosen carefully as too many data choices in a category may result in an explosion of test cases.

2.3 Activity Diagrams

As previously stated, our approach uses UML Activity Diagrams to model the logic captured by a use case. This type of diagram is ideal for our purposes because we require a method to describe the test case flow. In the test generation phase, an activity's text will become a test step. As is allowed in UML, activities can include or be refined by other activity diagrams. In these cases, the test generation will "flatten" the diagrams. The step containing the activity text will be replaced with the steps generated from the refined or included diagram.

An activity is considered refined when there is a sub-diagram associated to an activity, as permitted by the modeling environment. An activity includes another activity diagram when there is another diagram of the same name as the activity. The activity or a note anchored to the activity must have an <<include>> stereotype. Using notes in this situation encourages the reuse of diagrams: the same diagram can appear as the elaboration of activities at many points in the model.

All data objects relevant for a use case are modeled as test variables in the activity diagram. They are used to express the guard conditions in branches and to specify the data variations for test generation. The categories defined by classes (described above) provide the input data: the data choices in a category are the possible values of the variable. During test generation, the activity, which becomes a test step, requires one of the data choices of the category as an input in order to complete.

Before using test variables in branching conditions or for data variations, they must be defined and associated to an activity. A variable is defined by a note with the stereotype <<define>>, followed by the name of the category. The note has to be anchored to the activity where the data of the variable emerges. This is essential as the test generator must specify the value chosen for the variable/category at this particular step in the produced test cases.

Another stereotype, <<use>>, tells the generator to use a previously defined test variable rather than create a new instance of this variable. In other words, the same data choice is selected if the category has been used earlier in the path. If the variable has not been previously defined, then "use" has the same semantics of "define".

Once a variable has been defined, it can be used in the guard condition of a transition. Guard conditions must contain a comparison of a category to one of its data choices. The symbol "~" is used to designate equality and the keywords "not", "or" and "and" are permitted. For example, let us consider a category named "Shape" having the attributes "Circle", "Square" and "Rectangle". A transition located after this category having been defined as a variable could contain the expression "[Shape ~ Circle]". In the test generation phase, the test paths containing this transition would only have the data combinations that have selected circle as the data choice for shape.

In our approach, guard conditions of the transitions coming out of a decision point do not need to be mutually exclusive or even be deterministic. The lack of any guards on the transitions (from the same decision) indicates that each path is valid for any set of data values. If there is a guard on just one branch, the other branches are assumed to mean "otherwise". For our example above, a branch out of a decision point with the guard "[Shape ~ Circle]" implies that the other branches from the same point have the guard "[not (Shape ~ Circle)]". This helps simplify diagrams a great deal.

2.3.1 Example

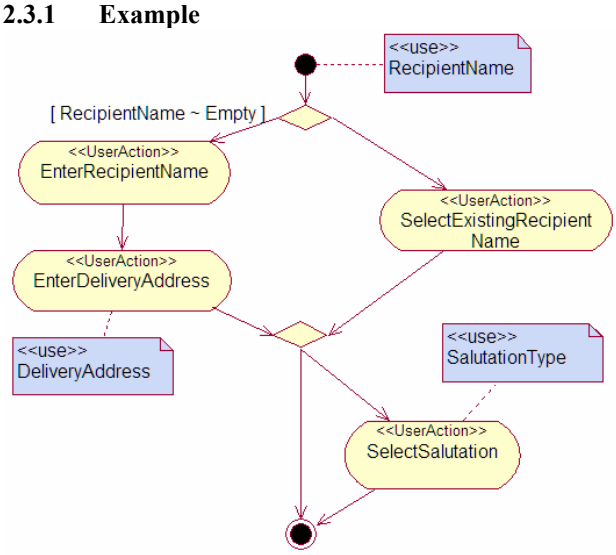

Figure 3: Specify Recipient Info Use Case

In this section we present a short example. Figure 3 depicts the use case "Specify Recipient Info" tab from the use case in Figure 1. In this tab, the graphical interface first requires the user to either select the recipient from an address book or to enter the name and address manually, then choose a salutation or not.

To model this, we introduce the variable "RecipientName" from the category of the same name (from Figure 2). Two of the data choices are names from the address book, which lead to the "UserAction" activity "SelectExistingRecipientName" and the third data choice, "Empty", which leads the path through the "UserAction" activities "EnterRecipientName" and "EnterDeliveryAddress".

The configuration options for the test generation, such as the path coverage criterion, will affect the number of test cases produced by this example. We discuss this further in sections 3.1 and 3.2.

3 Test Generation using TDE/UML

In this section, we show how the UML models are used as the basis for automatic test generation during system testing. From the UML activity diagram and the category data, test paths are created. To automate this, we use a tool built here at Siemens Corporate Research, the Test Development Environment using UML (TDE/UML). It works as a plug-in to many modeling environments such as Rational Rose, Borland's Together J, Argo/UML and our own tool, Eclipse (SCR) UML Diagram. TDE/UML exports relevant diagrams into an object representation, which allows it to be flexible regarding the multiple modeling tools. TDE/UML allows the user to specify many parameters of the test case generation. Due to length restrictions we present here just two of these options: the graph coverage criterion and the data coverage criterion.

3.1 Graph Coverage Setup

There are four different criterions that influence the graph coverage: Round Trip Criterion, Happy Path Criterion, All Paths Criterion and All Activities Criterion. This setup option determines the complexity of the path generation. The number of paths generated is only dependent on the chosen criteria. Later in the generation, some paths may be determined to be infeasible due to the data inputs and guard conditions.

For our example presented in Figure 3, the first three graph coverage criterions will yield the same result: four paths are generated. This is because our example is too simple: it does not have cycles or exception paths. However, the last criterion will produce fewer paths: only two paths are required to cover all the activities.

3.2 Data Coverage Setup

There are four different criterions that influence the data coverage: Sampling, Group Coverage Expression, Choice Coverage and Exhaustive Coverage. Exhaustive coverage, as well as undisciplined use of group coverage, can very easily generate an enormous amount of tests. Choice coverage ensures that each choice is present in at least

one test case. The number of test cases generated is not only dependent on the chosen data coverage criteria, but also on the graph coverage criterion. Changing any of these options will have a direct effect on the number of test cases the tool will generate.

4 Evaluating the Letter Wizard Example

In this section, we evaluate our approach using the Letter Wizard example. The most significant and time-consuming step in our approach is to define and annotate the activity diagram with test requirements. Preliminary investigations of related use cases and diagrams suggest that the number and complexity of the required annotations, including the one discussed here, is relatively small. This leaves the test designer focusing on the refinement of the diagrams and the definition of any data variations. A key prerequisite for this is a detailed understanding of the use case in order to express its associated business logic at the correct level of abstraction, at a level that allows the generation of test cases to be easily mapped to the implemented system.

Once the test design has been completed in terms of the activity, the additional effort to generate a set of textual test procedures or executable test scripts is minimal, at least for the example depicted here. The number of test cases generated for the Specify Recipient Info use case is presented in Table 1. As Choice and Exhaustive data coverage are more commonly used, only these are shown. Clearly, even for such a small example, the number of test cases can vary a great deal.

Table 1: Number of Test Cases – Specify Recipient Info

	Choice	Exhaustive
All Paths (4)	13	20
All Activities (2)	9	16

Table 2: Number of Test Cases – Letter Wizard

	Choice	Exhaustive
All Paths (288)	1550	41040
All Activities (10)	47	1632

We can generate the same table for the Letter Wizard use case (shown in Figure 1). As this is a more complex example, the number of test cases has exploded. This highlights the importance of having a concise model (using as many guards as possible) and the consequences of the chosen configuration. Obviously, the all paths criterion combined with exhaustive data coverage cannot be used in practice, unless the test step execution is completely automated, or if the modeled system is mission critical.

5 Conclusion

In this paper, we have described an on-going research project in system testing based on UML Models. The major concerns in our project are to improve the effectiveness and practicality on testing interactive systems and to address system testing in a "real world" scale. Test effectiveness is largely governed by the completeness, consistency, and accuracy of the supplied information and tester experience. A further issue is the degree of test automation being applied to support testing in large scale. For most organizations including Siemens, this can range from manually creating and executing a set of textual test steps for each regression test without any automation whatsoever, to a fully automated test suite with hundreds or even thousands of executable test scripts. The approach presented in this paper is clearly a step towards a comprehensive testing strategy for interactive systems.

6 References

[1] A. Abdurazik and J. Offutt, "*Using UML Collaboration Diagrams for Static Checking and Test Generation*", Proceedings of Third International Conference on the UML, pp. 385-395, Oct. 2000.

[2] L. C. Briand and Y. Labiche, "*A UML-Based Approach to System Testing*", Software and Systems Modeling, vol. 1 (1), pp. 10-42, 2002.

[3] A. Cavarra, J. Davies, T. Jeron, L. Mournier, A. Hartman and S. Olvovsky, "*Using UML for Automatic Test Generation*", Proceedings of ISSTA'2002, Aug. 2002.

[4] J. Hartmann, C. Imoberdorf, and M. Meisinger, "*UML-based Integration Testing*", Proceedings of ISSTA'2000, pp. 60-70, Aug. 2000.

[5] J. Offutt J. and A. Abdurazik., "*Generating Test Cases from UML Specifications*". Proceedings of 2nd International Confererence on UML'99, Oct. 1999.

[6] T. Ostrand, Marc J. Balcer: "*The Category-Partition Method for Specifying and Generating Functional Tests*", Comm. ACM vol.31, no.6, pp. 676-686 (1988).