Automation of GUI Testing Using a Model-driven Approach

Marlon Vieira, Johanne Leduc, Bill Hasling, Rajesh Subramanyan, Juergen Kazmeier
Siemens Corporate Research
755 College Road East, Princeton, NJ 08540
{marlon.vieira, johanne.leduc, bill.hasling, rajesh.subramanyan, Juergen. Kazmeier }@siemens.com

ABSTRACT

This paper describes an ongoing research on test case generation based on Unified Modeling Language (UML). The described approach builds on and combines existing techniques for data and graph coverage. It first uses 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 cases. 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 process of modeling UML activity diagrams, annotating them with test data requirements, and generating test scripts from the models is described. The goal of this paper is to illustrate the benefits of our model-based approach for improving automation on software testing. The approach is demonstrated and evaluated based on use cases developed for testing a graphical user interface (GUI).

Categories & Subject Descriptors:

Software Engineering, testing and Debugging, Testing tools

General Terms:

Verification, Reliability

Keywords:

UML, Model based testing, GUI Verification

1 Introduction

With testing activities accounting for a large part of the total effort of a software life cycle, incurred expenses are understandable. 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 also allow for automatic test case generation. Few existing approaches are based on Activity diagrams. These diagrams use only the control flow aspect of the modeled behavior. As far we know, ours is the only approach (research project) that generates 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 test cases (preparing for execution). This paper illustrates all those phases through an example and argues that the approach is useful particularly in the context of testing GUIs. Our research looks for robust ways to address the challenges of generating test cases from (UML) models with the right level of abstraction for automatic verification. The work reported here is part of a broader project on the development of practical approaches to apply model based testing in an industrial context.

Section 2 describes the modeling activities required by our approach. Our automation tool is described using an n example in section 3. Section 4 discusses the applicability of the approach. A common GUI, “the Letter Wizard” is the example used in this paper. This Microsoft Word™ 2003 application helps users to write and modify correspondence using a GUI.

2 Modeling Application Behavior

In this section, we describe the derivation of application 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 application and the actors who interact with the application 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 application and is the basis for testing the different functionalities and business rules described in the use cases specification.

The rationale for using Activity Diagrams for describing use cases is based on the assumption that activity diagrams express how functionalities can be exercised in terms of workflow and consequently are well

AST’06, May 23, 2006, Shanghai, China.

suited for creating "automatic test drivers" to verify those functionalities.

As mentioned before, modeling the application behavior, in our approach, means to define use cases and describe how to test those use cases with activity diagrams. In an ideal situation, use case modeling is performed by analysts during the requirements phase. Alternatively (or Failing which), the models can be created afterwards for testing. Basically, the models (i.e., use case + activity diagrams) can be used as a representation of which functionalities should be tested and a roadmap of how to test them. However, requirements models typically need to be enhanced to capture the information needed for generating useful test cases. We can classify these enhancements in two types. The first one is the refinement of activities on activity diagrams. This refinement generally leads to an improvement in the accuracy (e.g., decreasing the level of abstraction) of the described functionality, and consequently facilitating the efficiency of the test cases.

The second type of model enhancement is characterized by annotations on the activity diagrams. Our approach, support annotations by using custom stereotypes in notes anchored to a particular activity in the diagram. Those stereotypes represent additional test requirements and are typically added by the test designer. An example of annotation is data inputs, which are in the form of "categories" as defined in the data coverage section (section 2.2). These data inputs are combined with the generated test paths and primarily influence the test generation process in terms of the number of test cases generated, functional coverage attained and data selected for be used in each test case. This combination of activity diagram and data annotation enables our approach to reach a specific graph and data coverage during test case generation.

2.1 Use Cases

UML Use Case Diagrams are used to represent the functionality of the application from a top-down perspective. We assume that for test cases generation each use case, described in the Use Case Diagrams, has 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 GUI for the Microsoft "Letter Wizard", while Figure 2 depicts the Use Case Diagram for our example on testing the "Letter Wizard". The top level use case includes four other use cases, each representing a tab of the GUI.

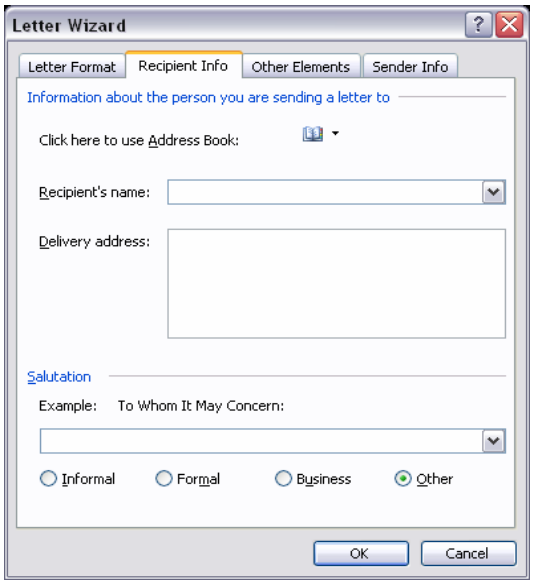


Figure 1: Letter Wizard GUI / Recipient Info Tab

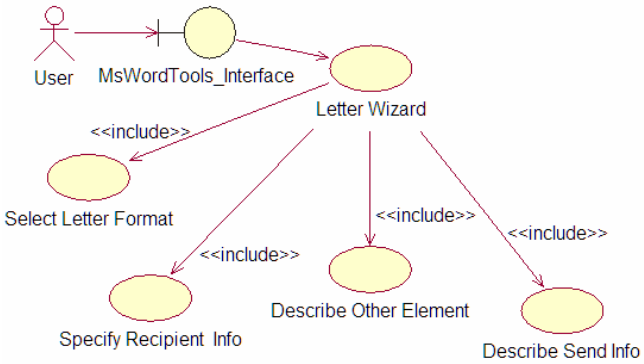


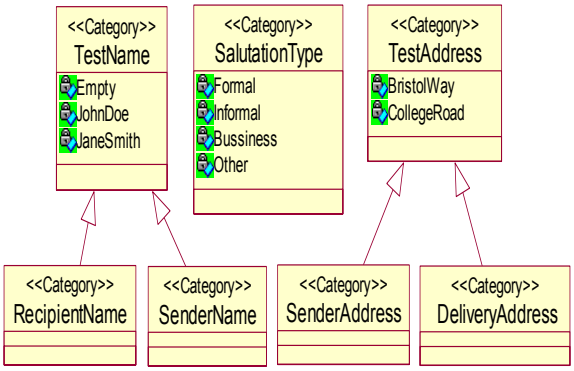
Figure 2: Example Use Case Diagram

2.2 Data Coverage

Underlying our UML-based testing approach is the category-partition method, which is a long term research at Siemens Corporate Research [6]. The category-partition method identifies behavioral equivalence classes within the structure of an application 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 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 are represented by classes with the stereotype <<Category>>. The data choices for each category are the attributes of the class. Figure 3 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.

Figure 3: Example Class Diagram



2.3 Activity Diagrams

As previously stated, our approach uses UML Activity Diagrams to model the functionality workflow defined 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 (i.e., enhance by refinement). 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. Another form of refinement is when an activity includes another use case, which has an activity diagram of its own. 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 (Figure 3) 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 (i.e., enhanced by an annotation). 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 syntax for describing guard condition follows the one defined for Siemens Test Script Language (TSL), which is a script language that supports the adoption of the category-partition method

Basically, TSL uses the symbol "~" to assign equality and the keywords "not", "or" and "and" to designate operations over those assignments. 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.

3 Example

In this section we present the activity diagram for the use case "Specify Recipient Info", depicted in Figure 4.

In the "Specify Recipient Info" 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.

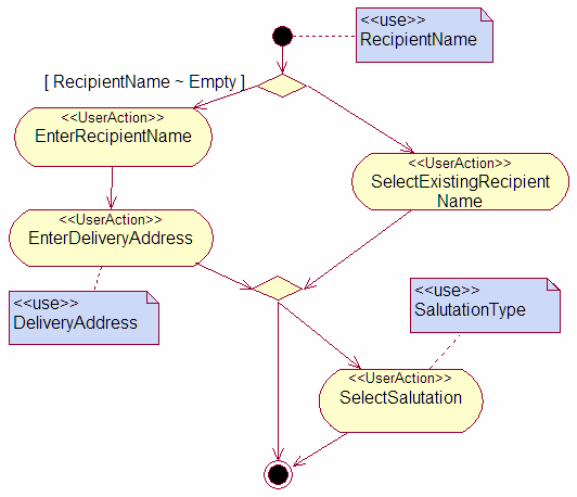


Figure 4: Specify Recipient Info Use Case

To model this, we introduce the variable "RecipientName" from the category of the same name (from Figure 3). 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 3.2 and 3.3

3.1 Test Generation

Before proceeding with a description of the test generation characteristics, we would like to emphasize that our approach generates a set of functional conformance tests. These test cases verify the compliance of what is implemented to what was modeled. It is assumed that the implementation behaves in a deterministic and externally controllable way. Otherwise, the generated test cases may be ineffective.

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 Editor. 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. We present in this paper two of these options: the graph coverage criterion and the data coverage criterion.

The output of the test generator is a set of XML-based files that can be formatted for presentation as a set of textual test procedures or executable test scripts based on the XSLT style sheet being used.

3.2 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 4, 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.

3.3 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.

3.4 Generating Test Cases for Sequences of Use Cases

Test cases can be generated from each use case to verify the described functionalities. In the "Letter Wizard" example, this means that each tab, e.g. "Select Letter Format", "Specify Recipient Info", "Describe Other Elements" or "Describe Sender Info" can be tested in isolation. However, it is also extremely important to verify sequences of those functionalities. That is, to specify one or more sequences that convey how combinations of functionalities can archive the end user's expectations. Those combinations can be represented as testing scenarios, which explicitly represent a set of use cases performed in a particular order.

In the case of GUI verification, testing scenarios normally starts with the user opening the GUI, continues with performing several use cases in sequence, and ends with system exit. Large systems typically involve enormous number of scenarios (sequences of functionalities). It is thus very important to wisely identify

important (e.g., common) scenarios for testing purpose. One approach is to identify one or more main scenarios, or "happy paths" and some alternative paths. These alternative paths may be exceptions (e.g., negative tests) or infrequently used options in the use case.

Figure 5 shows a testing scenario for the "letter Wizard" example. This model is a description for the top level use case ("Letter Wizard") and it includes the four other use cases. The depicted activity diagram captures a "testing scenario' which focuses on ordering the separate GUI functionalities in order to verify a complete interaction.

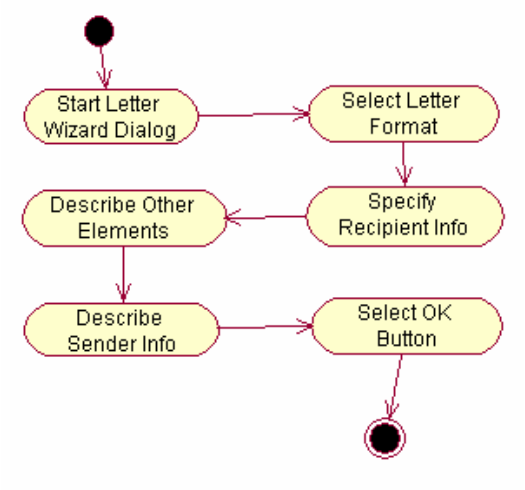


Figure 5 - Sequence of Use Cases

The discussion on the results of test case generation for the specified testing scenario will be carried out next.

3.5 Results from test case generation

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 represents the number of test cases generated for the "Letter Wizard" use case (testing scenario defined at sub-section 3.4).

Table 2: Number of Test Cases – Letter Wizard

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

It can be observed that the combination of functionalities typically leads to a large number of test cases. This highlights the importance of having a concise model (using as many guards as possible) and the consequences of the chosen test generation configuration. Obviously, the all paths criterion combined with exhaustive data coverage cannot be used in practice, unless the modeled application is mission critical.

Typically, a reasonable strategy for testing GUI is to use a stronger coverage for testing individual tabs or frames (e.g., All paths/Exhaustive data) and then use a weaker coverage (e.g., All activities/Choice or sampling data) for testing the overall interaction on the GUI .

Given some time/budget factors that most of the time is present on testing activity, a key issue is how to achieve the best possible coverage within available resources. Our model based approach allows testers to make that tradeoff by proving them the necessary information. Furthermore, it deals very well with scalability issues, since it make available ways to constrain the generated test cases and to chose the ones that are going to lead to the desired coverage.

4 Discussion

The most significant and time-consuming step in our approach is to define and annotate the UML diagrams with test requirements. Investigations recently performed suggest that with some training, the time necessary to create and annotate the UML diagrams for testing a specific application are not very different from manually creating and validating a comprehensive set of test scripts for testing the same application. This fact was observed in different projects where we are currently using the approach.

Our testing approach leaves the application testing group focusing on the creation and refinement of the models and the definition of data variations, instead of manually creating test scripts. This leads to a shift on the characteristics of the tasks performed by the test group. Therefore, some training is needed in order to understand the concept behind modeling and data annotations.

Another important observation is that the effort necessary to update the test cases against a later version of core specifications or the application implementation, is much reduced using our approach, since the modification involve to update the models and to regenerate the necessary test cases.

5 Related work

The use of UML for automatically generating test cases has been extensively studied in recent years. Many of these papers discuss work related to the automated test generation and execution for individual or subsystems of components. Our approach aligns more with those publications that discuss UML for the purposes of system

testing [1,2,3] and those that focus on generating tests for GUI-based systems, for example, Beer et. al. [5] describe the IDATG (Integrating Design and Automated Test Case Generation) environment for GUI testing. IDATG supports the generation of test cases based on both (1) a model describing a specific user task and (2) a model capturing the user interface behavior.

With respect to UML-based modeling for system testing purposes, Fröhlich and Link [1] discuss a method to automatically generate test cases from use cases. In their approach, a use case textual description is transformed into a UML statechart and then test cases are generated from that model. However, the use of statecharts in the context of representing system behavior is debatable – we found that activity diagrams reflected the user action/system response paradigm in a more natural way. Briand and Y. Labiche [3] propose the TOTEM system test methodology, which is based on the refinement of UML use cases into sequence (or collaboration) diagrams. It provides a good alternative approach to modeling with activity diagrams. Bertolino and Gnesi [2] present a methodology to manage the testing process of product lines. The methodology is based on annotation of textual use cases with category partition method information. Test cases are created based on those annotations. Their approach is similar to ours, since both approaches are based on category-partition method. The differences emerge with respect to two issues: a) we annotate the activity diagram instead of the textual use cases, and b) we make use of more precise structures based on TSL language, which leads to more consistent test scripts.

6 Conclusion

In this paper, we have described an on-going research project in application testing based on UML Models. The major concerns in our project are to improve the effectiveness and practicality of software testing and to address application testing in a "real world" scale. Test effectiveness is largely governed by the completeness, consistency, and accuracy of the supplied information and tester experience. Another 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.

Research is underway on some topics, but are not discussed in this paper. For instance, we aim to improve the test case generation with more detailed oracles. In the approach described in this paper the notion of a test passing (or failing) is related to the ability of the test driver to execute all specified test case steps successfully. We are investigating ways to improve the test cases with detailed verification points and arbiters. Another research issue being investigated is obtaining a more precise measurement technique for data coverage, particularly for improving the process of test data creation and utilization of the data during test script execution. Lastly, generation of executable test cases, for example, test cases in JUnit or CppUnit is being studied.

7 References

[1] P. Fröhlich, J. Link, "Automated Test Case Generation from Dynamic Models". In: Bertino, E. (Ed.): Proceedings of the ECOOP 2000 pp.472–491, 2000.

[2] A. Bertolino and S. Gnesi: "Use case-based testing of product lines". Proceedings of the ESEC / SIGSOFT FSE, 355-358, 2003.

[3] L. C. Briand and Y. Labiche, "*A UML-Based Approach to Application Testing*", Software and Applications 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] A. Beer, S. Mohacsi, and C. Stary: "IDATG: An Open Tool for Automated Testing of Interactive Software". Proceedings of the COMPSAC '98 - 22nd International Computer Software and Applications Conference, pages 470-475 August 19-21, 1998.

[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).