Innovations Syst Softw Eng (2005) 1: 12–24
DOI 10.1007/s11334-005-0006-0

ORIGINAL PAPER

Jean Hartmann · Marlon Vieira · Herbert Foster
Axel Ruder

A UML-based approach to system testing

Received: 6 November 2004 / Accepted: 6 January 2005 / Published online: 11 March 2005

Abstract This article describes an approach for automatically generating and executing system tests, which can be used to improve the validation of an application. Tests are automatically generated from behavioural models of the application using the unified modelling language (UML) and then executed using a suitable test execution environment. For this paper, we demonstrate our approach by means of an application that interacts with its users via a graphical user interface. Thus, we discuss the test- execution phase with respect to a commercial user interface (UI) or capture–replay tool.

In this article, we show how, in the first step, a test designer manually annotates the UML models, which may have been semiautomatically extracted from existing, textual-use case documentation, with test requirements. In the second step, the test- generation tool automatically creates a set of textual test procedures (test cases) or executable test scripts. In the third step, a test executor runs these against the system under test using a commercial UI testing tool.

The goal of the article is to illustrate the benefits of an automated, model-based approach for improving system test design, generation and execution. Details regarding the different tools are given.

Keywords Unified modelling language · System testing · Test generation · Functional testing · Use cases · Activity diagrams

J. Hartmann (✉)
Microsoft Corporation,
1 Microsoft Way,
Redmond, WA 98052, USA
Tel: +1-609-7343313
Fax: +1-609-7346565
E-mail: jeanhar@microsoft.com

M. Vieira · H. Foster · A. Ruder
Siemens Corporate Research,
755 College Road East,
Princeton, NJ 08540, USA
Tel: +1-609-7343361
Fax: +1-609-7346565

1 Introduction

System testing, which ensures the functional compliance of an application with its requirements, is a well-defined process within Siemens; however, in many cases, it remains a manual process. Test designers typically derive their test data, that is, their required system input and expected output information, from a variety of sources, including textual-use case specification and business-process rules. They then create a set of test procedures comprising individual test steps, which are executed manually by test executors against the system under test. Whenever an automated-test execution environment is available, these test executors are also responsible for translating these textual test procedures into executable test scripts. Our approach aims to automate and formalize the process of test design, generation and execution as much as possible, in order to deliver a more systematic and efficient system-testing process. Our approach encompasses the following tasks:

1. *Modelling system behaviour.* By modelling from scratch or semiautomatically converting existing textual-use case specifications into the appropriate UML models, we believe that the effectiveness of the test-design phase can be improved. By visually capturing the flow of interactions between the system and its users, it is possible to elicit, evolve and communicate a better, more complete test design. It enables test designers to identify and concisely document a greater variety of test scenarios than would be the case when writing complex, textual-use case descriptions.
2. *Generating test procedures.* Using the above explicit visual models of system behaviour, it is then much easier to manually or automatically create a set of test procedures, making the derivation of tests more systematic and efficient. Another benefit of our approach is that the notion of test adequacy or coverage with respect to the system functionality is addressed. Test designers now have a way to better quantify their testing efforts.

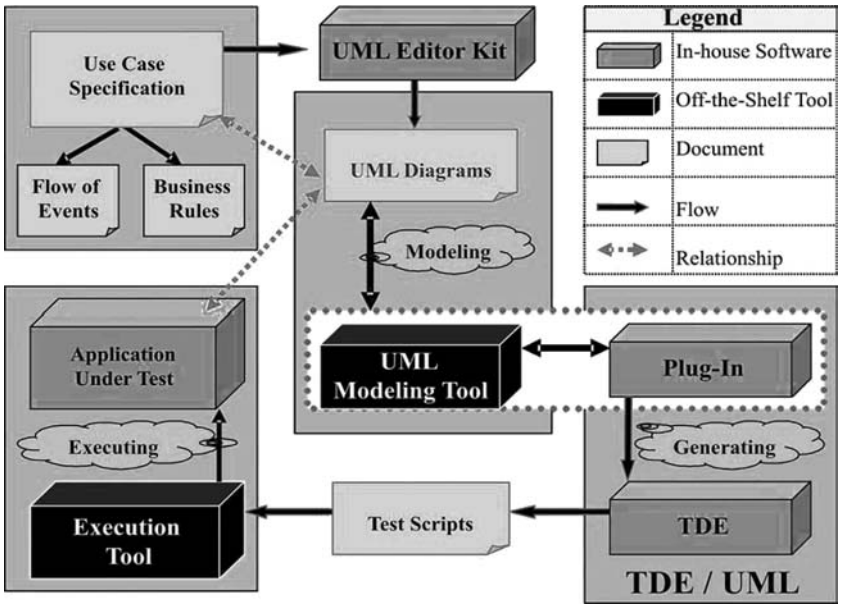

Fig. 1 Our approach to improve system testing

3. *Executing test scripts.* Automating test execution in itself helps to alleviate the error-prone and tedious task of regression testing; however, it is also a necessity in our approach, as the test-generation step may potentially result in a large number of executable test scripts being created. A key benefit of our approach is that it encourages test executors during initial user interface (UI) test capture to design and create a modular set of executable test scripts or test snippets, which later on also promotes script reuse and simplifies script maintenance. Once captured and parameterised, it is these snippets that are combined during the test-generation step to form complete test scripts.

Figure 1 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 semiautomatically convert an existing set of use case specifications, written in Microsoft Word, into models readable by Rational Rose, the UML visual modelling tool. Users then annotate and refine the UML diagrams in preparation for test generation. Our test generator, known as test development environment (TDE), which has been integrated (as a plug-in) with Rational Rose via the rose extensibility interface, is then executed in order to generate the required set of test cases in eXtensible markup language (XML). These test cases are then in turn converted to a set of executable test scripts by means of an appropriate extensible stylesheet language (XSL) template and executed using the UI test-execution tool of choice.

In Sect. 2, we describe our approach for modelling system behaviour using UML use cases and activity diagrams. Section 3 describes the various types of model annotations that can be added by a test designer before automatic test generation. In Sect. 4, we discuss the role and activities of a test executor in using commercial UI testing tools to prepare a library of test snippets. In Sects. 5 and 6, we provide an overview of the TDE/UML environment that realizes the approach and apply it to an example. Related work, conclusions and future work are presented in Sects. 7 and 8.

2 Modelling system behaviour in UML

In this section, we describe the use of UML use cases and activity diagrams in modelling the dynamic behaviour of systems. To better convey the concepts, we have illustrated the concept with an example. The example in Fig. 2 shows a use-case diagram for a set of related use cases from a web-based Siemens application. The diagram shows how the user—a hospital registrar—is able to create and cancel so-called patient encounters, that is, the hospital visits of a given patient.

In this paper, our focus will be on the *CancelEncounter* use case and its related set of use cases, *PrintArtifacts*, *ValidateEncounterDates* and *UpdateEncounter*. The relation between use cases is shown by the ≪include≫ stereotype.

2.1 Use-case specifications

While use-case diagrams are useful in showing the interaction between various individual use cases, use-case specifications and activity diagrams enable test designers to textually and visually capture the flow of control between the system and the user, respectively.

Use-case specifications are typically represented in tabular form and provide descriptive text for the individual user stimuli and system response (test) steps. This is done for both

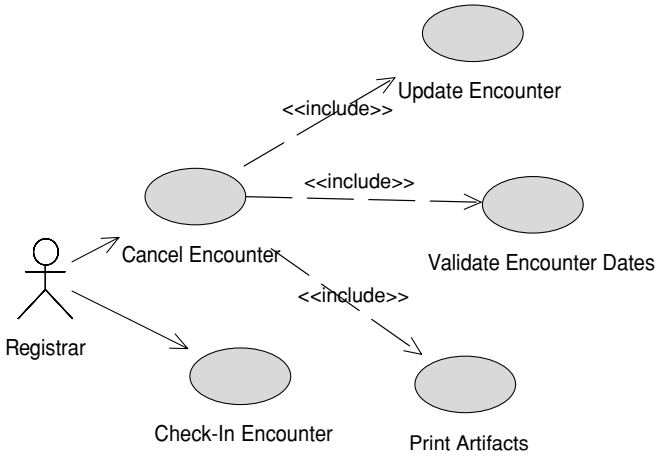

Fig. 2 Use-case diagram for *CancelEncounter* example

the main success scenarios and any alternative courses. The names of associated use cases are also mentioned, should this particular use case happen to traverse them during testing.

While the creation of textual-use case documents, especially when using document templates, is straightforward, these documents very quickly become unmanageable when considering complex use cases and test scenarios, which include multiple, nested alternative flows. As complexity rises, test designers may overlook important test scenarios when deriving their test procedures from such documents. A much more concise and compelling way of eliciting, evolving and communicating such test scenarios is by means of activity diagrams. Thus, we provided the UML Editor Kit to help users quickly convert the existing documents into UML activity diagrams.

2.2 Activity diagrams

Activity diagrams are very well suited to reflect the user input and System-response paradigm by being partitioned into so-called swim lanes. The first pair of lanes shows the typical test scenario, flow of interaction or happy path. The remaining lane depicts the alternate courses. Figure 3 shows a completely annotated activity diagram for the *CancelEncounter* use case—just as it is required for test generation. We now describe the different types of annotations that must be made to a standard activity diagram by a test designer.

The test requirements described below represent diagram annotations that primarily influence the test-generation process in terms of the number of test cases generated and the functional coverage attained. Many of them are optional and typically do not have to be added to the diagram; we indicate which ones these are.

2.2.1 Adding the test requirements

Each activity, shown by an elliptical box, 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. In our example, these stereotypes were added automatically as part of the conversion process from the textual-use case specification.

Depending on the complexity of the system under test, it may be necessary to specify expressions in the guard conditions which govern conditional flows[1]. 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. In Fig. 3, for example, the activity *FilloutCancellationForm* requires a test executor to input a valid calendar date for the patient encounter that is to be cancelled. Thus, this activity requires a test designer to define a variable for each of the possible outcomes. For example, the variable *InvalidDate* is defined, so that the system can be checked for its response to an invalid date for the encounter. This is important because the test generator has to specify the value chosen for the variable at this particular step in the generated test cases.

Diamond boxes represent conditional flows of interaction described by the diagram. These points typically include guard conditions that guide branch selection; those branches not labeled with expressions may be considered as default branches in the flow. Depending on the complexity of the system under test, it *may* be necessary for these guard conditions to be specified as expressions that can be recognised by the test generator and thus must be written using test-specification language (TSL) syntax[2]. The variables used in these expressions must be defined as described earlier—using the ≪*define*≫ stereotype. If no expressions are specified, then one branch typically evaluates to TRUE and the other to FALSE.

Because the category-partition method is the underlying test generation technique in our approach, variable ranges are partitioned into equivalence classes. By default, every variable is considered to be Boolean and therefore partitioned into two choices, TRUE and FALSE.

For variables of type other than Boolean, the variable ranges have to be explicitly specified using a ≪partitions≫ stereotype and depicted in the diagram as a text label. For example, *ExistingServicesProfileOption*, which is defined and used as a variable during the *InvokeCancellation* activity, requires partitioning into two so-called choices: *warning* and *error*. Another example from the related *PrintArtifacts* use case specifies that the system shall display on the screen, a prompt for the user to enter the number of receipts to be printed for the cancelled encounter. Based on data from the

[1] These expressions must usually be derived manually from the textual documents, as they are often implied. If they could, however, be specified explicitly in these documents, then they can be automatically extracted and placed in the activity diagram.

[2] They could be specified in OCL, the Object Constraint Language, but we would need to translate these into TSL conditions.

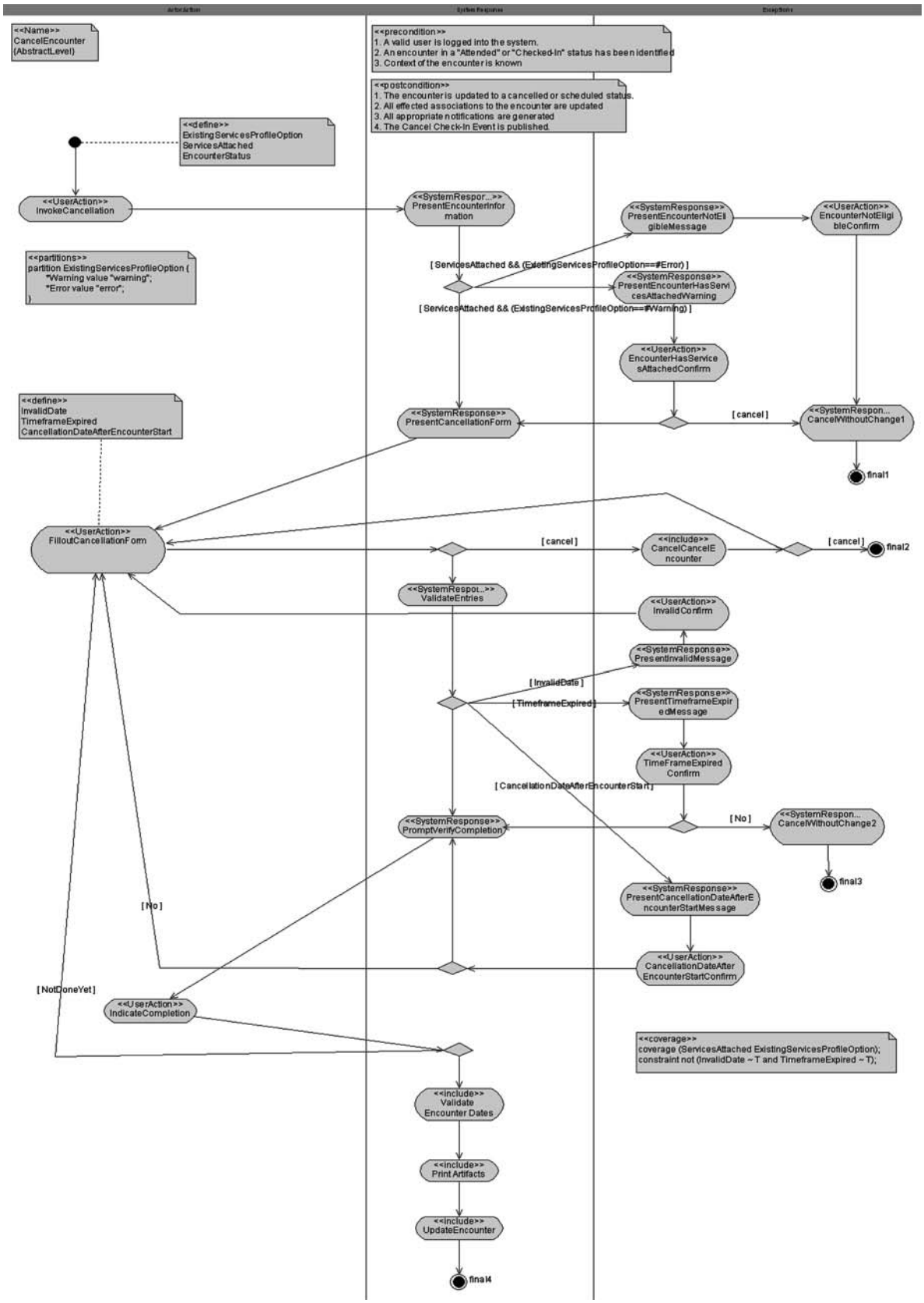

Fig. 3 Activity diagram for *CancelEncounter* example

use-case documentation or discussions, it is determined that the partition *EnterInput* includes the choices: **zero value 0; *lownumber value select (1,. . . ,99); *highnumber value 100*.

While, by default, the coverage criterion being applied is transition coverage, the test designer can influence the test-generation process by specifying optional coverage requirements using the ≪*coverage*≫ stereotype in a text label. The coverage statements must be specified as TSL coverage expressions. In Fig. 3, for example, the coverage expression shown ensures that test cases are generated to cover all combinations of the two specified variables; however, the constraint ensures that no test case is generated where the *InvalidDate* and *TimeFrameExpired* field entries are TRUE.

At this time, any precondition and postcondition specified in the use- case documentation are simply marked with the appropriate stereotype during conversion and placed as a text label in the UML activity diagram. The test generator does not process them at this time but includes them in the set of test cases as a prologue.

2.2.2 Refining the diagram

Test designers may wish to refine their activity diagrams by creating subdiagrams for particular activities to describe them in more detail to the test executor or test generator. Figure 4 shows this for the user activity *FilloutCancellationForm* of Fig. 3.

Because, during the test generation process, the original activity is replaced by its subdiagram, all variables that are defined by this activity also have to be defined in the subdiagram or using a ≪*refine*≫ stereotype. In the given example, the variable *InvalidDate* is refined in the subdiagram. Every occurrence of *InvalidDate* will thus be substituted by the expression *InvalidCancelDate* || *InvalidCheckInDate* during test generation.

The test designer may also wish to influence the depth to which subdiagrams are expanded. This provides a mechanism for generating test cases with different degrees of abstraction from the same model. Using a ≪*name*≫ stereotype as a test label, test designers can specify the name of the activity diagram and declare the level of abstraction as a UML property. Subsequently, the test generator does not expand and process any subdiagrams at levels below the one specified.

3 Test generation

Before proceeding with a description of the test generation and execution steps, we would like to emphasise the following:

- Our approach generates a set of conformance tests. These test cases aim to ensure the compliance of the system specification with the resulting implementation.
- It is assumed that the implementation behaves in a deterministic and externally controllable way. Otherwise, the generated test cases may not be meaningful.

3.1 Category-partition method

For test generation, we use the TDE, a product developed at Siemens corporate research [1]. The TDE processes a *test design* written in the test-specification language (TSL). This language is based on the category-partition method, which identifies behavioural equivalence classes within the structure of a system under test [2].

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.

A TSL test design is now created from the activity diagram by mapping its activities and transitions to TSL partitions and choices. It is important to realise 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.

3.2 Generation procedure

A recursive, directed graph is built by the TDE that has a root category/partition and contains all the different paths of choices to plain data choices. This graph may contain cycles depending on the choice definitions and is equivalent to the graph of the global state machine. A test frame, that is, test case, is one instance of the initial data category or partition, that is, one possible path from the root to a leaf of the (potentially infinite) reachability tree for the graph.

An instantiation of a category or partition is a random selection of a choice from the possible set of choices defined for that category/partition. In the case of a category, the same choice is selected for every instance of a test frame. This restricts the branching possibilities of the graph. With a partition, however, a new choice is selected at random with every new instantiation. This allows full branching within the graph and significantly influences test data generation. The contents of a test case consist of all data values associated with the edges along a path in the graph.

3.3 Coverage requirements

The TSL language provides two types of coverage requirements:

- *Generative requirements* control which test cases are instantiated. If no generative test requirements are defined, no test frames are created. For example, coverage

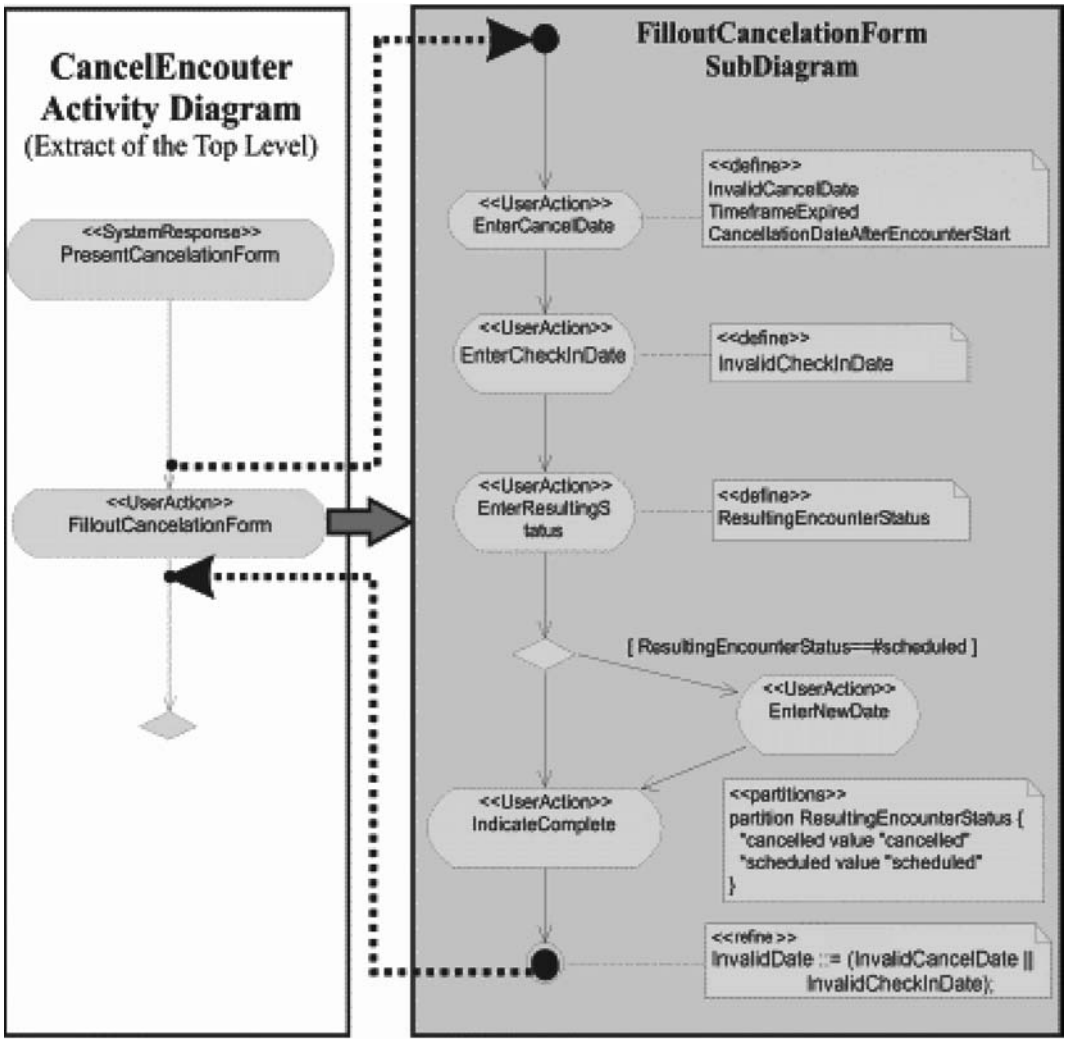

Fig. 4 Subdiagram for *FillOutCancellationForm*

statements can be defined for categories, partitions and choices.

- *Constraining requirements* cause TDE to omit certain generated test cases. For example, there are maximum coverage definitions, rule-based constraints for category/partition instantiation combinations, instantiation preconditions and instantiation depth limitations. Such test requirements can be defined globally within a TSL test design or attached to individual categories, partitions or choices.
- TDE creates test cases in order to satisfy all specified coverage requirements. By default, the coverage criterion is that all transitions within an activity diagram must be traversed at least once.

3.3.1 Included use cases

Included use cases represent related use cases and their associated activity diagrams, for example, *PrintArtifacts* and *ValidateEncounterDates*. These must be incorporated into the test- generation process for *CancelEncounter*. This is achieved in a manner similar to the processing of subdiagrams, but with one significant difference— because included use cases may themselves contain further use cases and so forth, this could potentially lead to a large number of test cases being generated, given the above default coverage criterion. Also, covering the branches for a set of included use cases is not really necessary because it is assumed that separate sets of test cases will be generated for them anyway. Therefore, the current default behaviour of the test generator is such that it only expands the main success scenario or happy path of the included use cases. It is possible, however, to modify this behaviour to enable test designers to generate test cases to traverse the branches of the target activity diagram and those of its included use cases. Also, the test designer can limit the level of use-case inclusion as described in Sect. 2.2.2 for subdiagramming.

3.4 Generated output

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 XSL style sheet being used. Figure 6 in Sect. 6.2 depicts a textual test procedure for one test sequence of the *CancelEncounter* diagram.

4 Test execution

In this section, we describe a test execution process that assumes automation and focus on describing how a structured approach to capturing, modularising and parameterising a representative set of executable test scripts can benefit regression testing, especially in conjunction with our approach.

4.1 Initial script capture

Using the set of textual test procedures, which were generated earlier, the test executor can now start with the capture of a representative set of test scripts using a commercial UI (capture–replay) tool. More precisely, the goal is to record individual test steps, each corresponding to a user activity in the activity diagram. Steps need to be recorded until all transitions in the diagram have been traversed.

While path verification is being handled in each test step by checking whether the proper graphical window is being displayed, additional data- verification test steps may need to be created in order to check that the data fields contain correct values. Alternatively, the verification may be performed as part of the original set of test steps. Some test tools allow the capture of so-called checkpoints that can refer to this data. Other tools only capture the data on the screen and require data verification to be performed programmatically by the test executor.

As our approach focuses on three-tiered applications, database verification is an important consideration-it needs to be performed at the end of each test sequence after database changes have been committed. After the test sequence has been captured and the GUI state is at the Home page (see Section 4.2), additional steps are recorded to check the state of the database. If the relevant database state can be observed in the GUI being tested, those screens are navigated to, and the data are verified. Alternatively, if a database checking tool is available, scripts would be written to verify the fields in question, and these scripts would be called.

4.2 Script execution and refinement

At this stage, a trade-off needs to be considered with respect to our approach. If budget or time is a constraining factor, a test executor may wish to either just maintain the representative set of test scripts and models or enhance the existing set of scripts by recording additional scripts, which vary only in the data values being entered in the application's input fields.

Alternatively, our approach would encourage the test executor to invest additional effort in crafting a library of test snippets from the above representative set of test scripts to allow the test generator to automatically create those additional tests. Our approach requires the individual test steps to be refined—generalised or parameterised—so that they are independently callable from an automatically generated test script and driven using their arguments or the data structures that reference those arguments. In addition, the approach requires the test executor to customise the XSL style sheets in order for the test generator to generate the executable test scripts.

Good testing practice dictates that there be no dependencies between test cases. Because the order of test cases that the test generator creates is random, there should be no dependency between test cases. For example, if one test case cancels an encounter and another stops the cancellation, the encounters for the two test cases must be separate encounters.

In order to prevent test dependencies, each test should leave the GUI in the same state as at the start of the test. Typically, this means that each test should return to the Home Page when validating web-based applications or should exit the GUI.

The process of creating such a library requires close cooperation between test designers and test executors. Differences between the model and implementation would indicate potential bugs.

Figure 7 in Sect. 6.3 shows a generated test script with calls to the individual test step functions.

4.3 Script maintenance and regression testing

One of the biggest concerns with testing GUI applications automatically is that script maintenance is difficult when the underlying application logic or GUI layout changes. Often, the same test step is captured in several different contexts and thus must be maintained in all of them. Sometimes paths change and a whole sequence of test steps needs to be identified and captured.

In our approach, there can be two outcomes. If the test executor has maintained the representative set of test scripts and models as well as added test scripts containing data variations, then, depending on the type of change to the application, the test designer would need to update the models and can at least regenerate the *textual test procedures* for the test executor. Some of these test procedures would map to existing test scripts, but the test executor would have to perform a manual comparison. In other cases, scripts would have to be updated by the test executor with entire new test sequences.

The more advanced solution involved significant work going into generalising each test step and making it reusable. This expense can now be recouped. For some changes, the test designer would similarly update the models, but instead be able to regenerate the *executable test scripts*, requiring minimal intervention from the test executor. Other types of

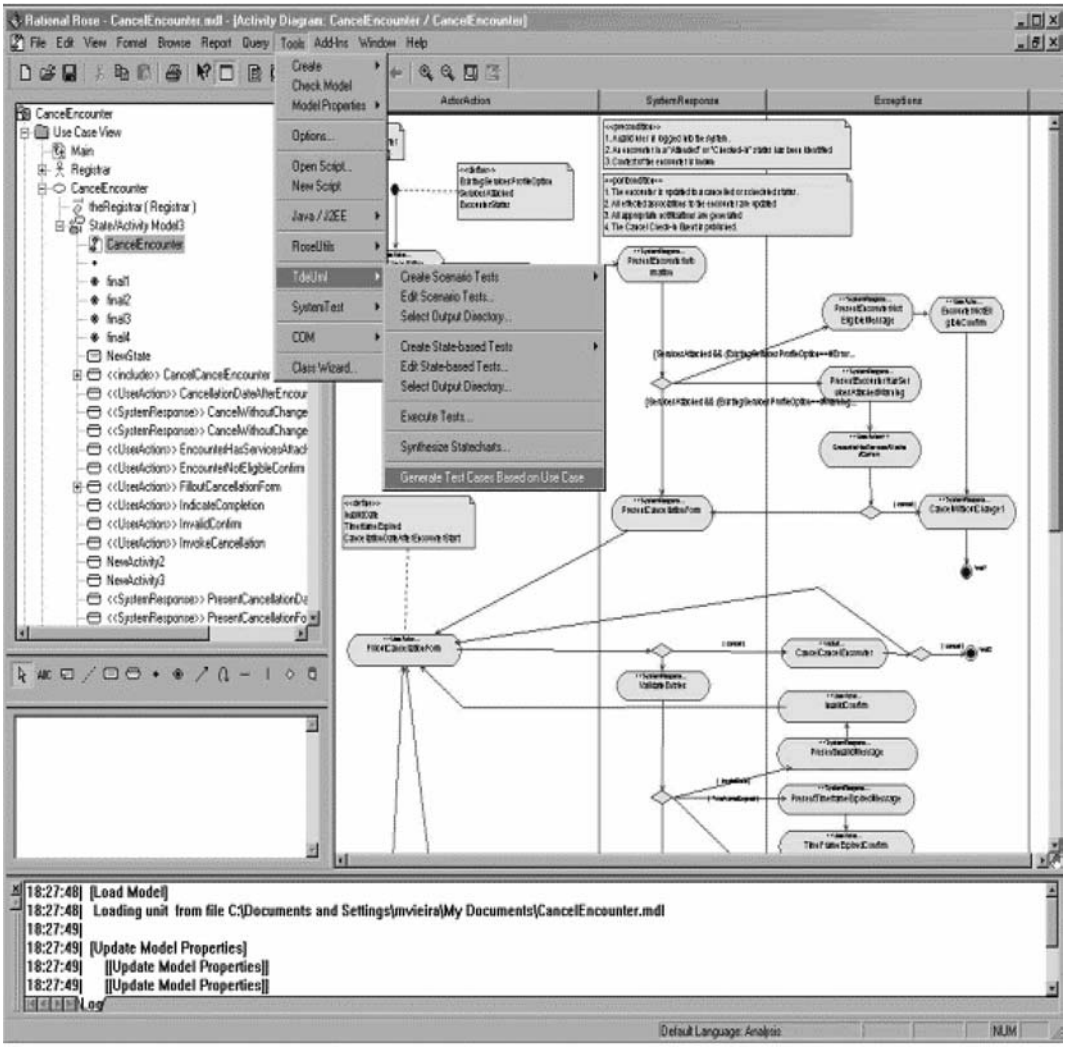

Fig. 5 Generating system tests from within Rational Rose

changes would only require new *test steps* rather than entire new sequences to be recorded and parameterised.

5 Implementation

For this approach, the T*n*T environment that was originally developed at Siemens Corporate Research to perform UML-based unit and integration test generation for components was enhanced [3]. It is now capable of not only generating test cases from activity diagrams but also state charts and sequence diagrams. The environment has been updated to work with the Rational Rose 2002 UML modelling tool and the Compuware TestPartner 3.0 UI test execution tool, although other UI testing tools can be easily integrated. Figure 5 shows how test-case generation can be initiated from within Rational Rose.

In this section, we describe our implementation strategy.

5.1 UML editor kit

In order for Siemens groups, with large amounts of existing textual-use case specifications, to take advantage of our approach and quickly generate the basic UML activity diagrams, Siemens Corporate Research developed the UML Editor Kit. This kit allows test designers to semiautomatically convert their existing textual-use case specifications into activity diagrams, which may then be annotated using the test requirements described in Sect. 2.2.1. *Semiautomatically* refers to the process of users displaying their Microsoft Word use case documents using the Editor Kit, annotating them in such a way that the test cases are clearly delimited and then having the tool automatically parse those annotated documents to convert them into XML metadata interchange (XMI) files for later input into Rational Rose. While this may be a considerable amount of effort, it would still be more efficient than creating corresponding UML activity diagrams from scratch.

5.2 TDE/UML

TDE/UML accesses the Rose application through a Microsoft COM interface. In fact, our application implements a COM server, that is, a COM component waiting for events. We implemented TDE/UML in Java and used Microsoft's Visual J++ to interface to Rose, as it can generate Java classes for a given COM interface. Each class and interface of the Rose object model can thus be represented as a Java class; data types are converted and are consistent. The Rose applications export administrative objects as well as model objects, which represent the underlying Rose repository.

Rose also provides an extensibility interface (REI) to integrate external tools known as *add-ins*. A new tool, such as TDE/UML, can be installed within the Rose application as an add-in and invoked via the Rose *Tool* menu. Upon invocation, the current Rose object model is imported, including the necessary activity diagrams, processed using the techniques described in previous sections, and the files needed for test generation and test execution are generated.

5.3 UI testing (capture–replay) tool

Our selection criteria for a commercial UI or capture–replay tool included the following:

- The ability to record and play back user actions based on logical objects, not screen or window positions. Most tools fulfil this criterion.
- The availability of a powerful test scripting language, preferably a complete programming language, such as Microsoft Visual Basic. The generated test scripts needed to call sequences of user actions as methods or functions.
- The ability to encapsulate data for tests to hide details concerning the GUI and database.

6 Evaluating the example

In this section, we evaluate our approach using the *CancelEncounter* example. We briefly discuss how, in practice, each step of our approach is applied, provide sample test scripts, and evaluate it against the current manual system testing approach used at Siemens.

A more extensive evaluation of our approach is currently being discussed within Siemens, so that results from that experimentation are not yet available. We are, however, aiming to examine issues such as the fault- detection capabilities and cardinality of the generated test cases; the time and effort saved by the automated design, generation and execution; as well as the complexity of the overall approach.

We worked closely with our colleagues in the business unit, so that, for the evaluation, it can be assumed that we had the equivalent domain knowledge, enabling us to design, generate and execute tests using our own techniques and tools.

6.1 Annotating the activity diagram

The most significant and time-consuming step in our approach is defining and annotating 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. In many cases, the test designer can rely on the default values used by the test generator for coverage and conditional expressions. This leaves the test designer focusing mainly on the refinement of the diagrams and the definition of any data variations. A key prerequisite for this is the need for a detailed understanding of the use case in order to express its associated business logic at the correct level of abstraction, that is, at a level that allows the generation of test cases to be easily mapped to the implemented system.

6.2 Generating the test cases

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; however, we do hope to be able to better quantify this once an empirical study is underway.

While the test-generation step took mere seconds to complete for the *CancelEncounter* example, it is worthwhile noting that a significant portion of that time was taken up with computing the test cases for the main success scenarios of each adjoining use cases and then substituting them in the TSL test design for *CancelEncounter*.

Figure 6 shows a sample textual test procedure with the choices as well as user actions (required inputs) and system outputs (expected responses) bolded. The test procedure also contains a description of the preconditions and postconditions that the system needs to be in before the start and after the end of the test. Currently, TDE does not implement any further processing for preconditions and postconditions, that is, these conditions are simply displayed in the textual test procedures but not in the automated scripts.

A further point of interest is the use of choices to define the initial state of the system before testing and also influence test-case execution. With reference to Fig. 3 and stepping through Fig. 6, the choices influence execution of the main success scenario until the *ValidateEntries* activity. Then, an alternative path through the *PresentTimeframeExpiredMessage* activity is chosen as the variable *TimeframeExpired* is TRUE. Now, the flow returns to the main success scenario in the *PromptVerifyCompletion* activity and so forth, until it finally completes the verification when cancelling a patient encounter. Note that here the *FilloutCancellationForm* activity (A) is represented by its refinement in Fig. 4. Note that, while the main success scenario sequence for use cases *Validate_Encounter_Dates* and *PrintArtifacts* are not explicitly described here (B), the use case *UpdateEncounter* is described as a sequence of outputs (C), that is, expected system responses after an encounter is cancelled.

```
Test name="CancelEncounter_SystemTest_5"
  Preconditions
  {
     1.  A valid user is logged into the system.
     2.  An encounter status has been identified
     3.  Context of the encounter is known
  }
  Choice "EncounterStatus" = "Attended"
  Choice "ServicesAttached" = "False"
  Choice "ExistingServicesProfileOption" = "error"
    UserAction     "InvokeCancellation"
    SystemResponse "PresentEncounterInformation"
    SystemResponse "PresentCancellationForm"
    UserAction     "EnterCancelDate"
  Choice "CancellationDateAfterEncounterStart"="F"
  Choice "TimeframeExpired"="T"
  Choice "InvalidDate"="F"
(A)
    UserAction     "EnterCancellationReason"
    UserAction     "EnterResultingStatus"
  Choice "ResultingStatus" ="Scheduled"
    UserAction     "EnterNewDate"
    UserAction     "IndicateComplete"
  Choice "IndicateComplete_cancel" = "F"
    SystemResponse "ValidateEntries"
    SystemResponse "PresentTimeframeExpiredMessage"
    UserAction     "TimeFrameExpiredConfirm"
  Choice "TimeFrameExpiredConfirm_No" = "F"
    SystemResponse "PromptVerifyCompletion"
    UserAction     "IndicateCompletion"
  Choice "IndicateCompletion_NotDoneYet" = "F"
(B) SystemResponse "Validate_Encounter_Dates"
    SystemResponse "Print_Artifacts"
    SystemResponse "ValueCancellationType"
    SystemResponse "ManageSecEncounterLinksNecessary"
(C) SystemResponse "UpdateEncounterWithAssociations"
    SystemResponse "NotifyEffectedApplications"
    SystemResponse "GenerateCancellationNoticeDocument"
    SystemResponse "NotifyInterestedPartiesOfCheckInCancellation"
    SystemResponse "NotifyActorCancellationComplete"
Postcondition
{
     1.  The encounter is updated to a cancelled or
         scheduled status.
     2.  All effected associations to the encounter are
         updated
     3.  All appropriate notifications are generated
     4.  The Cancel Check-In Event is published.
}
```

Fig. 6 Sample textual test procedure

6.3 Executing test cases

As discussed in Sect. 4.2 and now shown here using the example, the individual test and verification steps have been codified using the scripting language of the capture/replay tool as *TestStep()* and *VerifyStep()* functions, respectively. Essentially, the function *TestStep()* simulates the user action and the function *VerifyStep()* verifies the system response is consistent with what is specified; in other words, it acts as an oracle.

The choices, that is, the data values used to initialise the context for the test, are supplied via the *SetTestParm()* function. For example, in order for the generated test case to be able to cancel an encounter, a specific encounter first needs to exist. Using the choices, *EncounterStatus* and *ServicesAttached*, a specific encounter from the test database is recalled that can then be cancelled during the execution of the test case. Those choices ensure that the encounter is classified as attended and cannot have services attached to it. (For a sample executable test script, see Fig. 7.)

6.4 Analysis

An informal evaluation was conducted pending our discussions concerning a detailed evaluation with the Siemens

```
Function RunTests ()
 StartRun 'Must be the first executable statement
  StartTest "CancelEncounter_SystemTest_5"
   SetTestParm  "EncounterStatus" , "Attended"
   SetTestParm  "ServicesAttached" ,"False"
   SetTestParm  "ExistingServicesProfileOption","error"
      RunStep     "InvokeCancellation",""
      VerifyStep "PresentEncounterInformation", ""
      VerifyStep "PresentCancellationForm", ""
      RunStep     "EnterCancelDate", ""
   SetTestParm   "CancellationDateAfterEncounterStart", "F"
   SetTestParm   "TimeframeExpired" , "T"
   SetTestParm   "InvalidDate", "F"
      RunStep     "EnterCancellationReason",""
      RunStep     "EnterResultingStatus",""
   SetTestParm   "ResultingStatus" , "Scheduled"
      RunStep     "EnterNewDate" ,""
      RunStep     "IndicateComplete",""
   SetTestParm   "IndicateComplete_cancel" , "F"
      VerifyStep "ValidateEntries" , ""
      VerifyStep "PresentTimeframeExpiredMessage", ""
      RunStep     "TimeFrameExpiredConfirm" ,""
   SetTestParm   "TimeFrameExpiredConfirm_No" , "F"
      VerifyStep "PromptVerifyCompletion" ,""
      RunStep     "IndicateCompletion",""
   SetTestParm   "IndicateCompletion_NotDoneYet" , "F"
     VerifyStep "Validate_Encounter_Dates" ,""
      VerifyStep "Print_Artifacts" ,""
      VerifyStep "ValueCancellationType" ,""
      VerifyStep "ManageSecEncounterLinksNecessary" ,""
      VerifyStep "UpdateEncounterWithAssociations" ,""
      VerifyStep "NotifyEffectedApplications" ,""
      VerifyStep "GenerateCancellationNoticeDocument",""
      VerifyStep "NotifyInterestedPartiesOfCheckInCancellation" ,""
      VerifyStep "NotifyActorCancellationComplete" ,""
    RunStep "Logoff",""
   EndTest
 End Run
End Function
```

Fig. 7 Sample executable test script

business unit. The goal was at least to compare, for this example, the effort required for modelling, annotating, generating and *manually* executing test cases versus the existing manual approach.

The results showed that the manual effort required **eight** staff-days to specify and refine 10 test procedures derived from the textual-use case specifications and business rules. A further **10** staff-days were required to execute and verify these tests. This resulted in a total effort of 18 staff-days to validate *CancelEncounter*.

In comparison, our approach requires **three** staff-days to define, annotate and refine the corresponding activity diagram, which had been semiautomatically derived from the textual-use case specification. With test generation time being negligible, the manual execution of the resulting five test procedures took a further 5 days. This resulted in a total effort of 8 days to validate *CancelEncounter*.

The manual approach concentrated on testing the described informal business rules rather than the more formal definition reflected in the activity diagram, which was used by our approach. According to the Siemens test designers, this accounted for the larger number of test procedures. Nevertheless, a detailed re-examination of our generated test procedures showed that all informal business rules had been satisfied to the same degree as the manual approach. In addition, the test designers could now rely on a notion of functional coverage whereby every activity and transition was covered.

While our approach ensures the validation of individual use cases for a system, it also supports the validation of multiple-use cases that interact. This enhances the test designers' confidence that the functionalities are working together as intended.

With respect to fault-detection capability, we could not draw any Conclusion, as neither approach detected any further bugs for a given release—the *CancelEncounter* feature was one of the more mature features in the system.

7 Related work

The use of UML for automatically generating test cases has been extensively studied in recent years [3–7]. Many of these papers discuss work related to the automated test generation and execution for individual or subsystems of components. This work is more applicable to unit and integration testing.

Our approach aligns more with those publications that discuss UML for the purposes of system testing [8–10] and those that focus on generating tests for GUI-based systems [11,12]. This includes previous work conducted at Siemens Corporate Research with similar, but less formal, goals [13].

For example, Beer et al. [11] describe the integrating design and automated test case generation (IDATG) 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 behaviour. The tool also supports test-case execution based on commercial capture–replay tools similar to the one used in this paper. A similar environment, developed by Menon [12,14], consisted of test generation, coverage, execution, oracle creation and regression analysis features.

A key difference between these last two approaches and our approach is That, in our case, we aim to validate system functionalities and business rules via the GUI and do not aim to test or cover the user interface functionalities as such. Therefore, we needed not only to develop a behavioural model (encompassing control information) but also to apply a strong data model specification technique, which is achieved by using TDE, for test-case generation. A further difference is that our approach uses UML behavioural models for test generation, while the other two approaches define their own custom modelling notations.

With respect to UML-based modelling for system-testing purposes, Fröhlich and Link [10] discuss a method to automatically generate test cases from use cases. In their approach, a use-case textual description is transformed into a UML state chart and then test cases are generated from that model; however, the use of state charts in the context of representing system behaviour is debatable—we found that activity diagrams reflected the user-action–system-response paradigm in a more natural way. Briand and Labiche [9] 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 modelling with activity diagrams.

Bertolino and Gnesi [8] 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 because both approaches are based on the category-partition method. The differences emerge with respect to two issues: (1) we annotate the activity diagram instead of the textual-use cases and (2) we make use of more precise structures based on TSL language, which leads to more consistent test scripts.

8 Conclusions and future work

In this article, we have described an on-going research and development project at Siemens Corporate Research in which UML-based models are being used to improve system testing of interactive applications. We have described the different aspects of our approach, discussed their implementation and evaluated it using an example from a web-based Siemens IT application.

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 technique.

Several topics for future research have been identified during this work. First, we aim to improve the performance of the test-case generation step and the reliability of the scripts executed. Second, the implementation of a more precise measurement technique for data coverage is important, particularly when we want to improve the process of test data creation and utilisation of the data during test script execution. A third step is to make use of specific test scenarios to verify the system's business rules. These rules specify constraints on the system and they can be efficiently verified through detailed usage scenarios. Finally, we aim to implement better mechanisms for further composing the activity diagrams and simplifying them as well as obtaining more concise models for system testing.

Acknowledgements We would like to thank Dr. Jürgen Kazmeier, the Head of the Software Engineering Department at Siemens Corporate Research, as well as Professor Manfred Broy and Michael Meisinger at the Technical University, Munich. We would also like to express our thanks to Daphne Thomas, Frank Heer, Seth Synder and Abraham Shenoy at the Siemens business group for their cooperation.

References

1. Balcer M, Hasling W, Ostrand T (1990) Automatic generation of test scripts from formal test specifications. In: Proceedings of ACM SIGSOFT'890–third symposium on software testing, verification, and analysis (TAVS-3). ACM Press, New York, pp 257–71
2. Ostrand T, Balcer MJ (1988) The category-partition method for specifying and generating functional tests. CACM 31(6):676–686
3. Hartmann J, Imoberdorf C, Meisinger M (2000) UML-based integration testing. In: Proceedings of ISSTA 2000, pp 60–70

4. Abdurazik A, Offutt J (2000) Using UML collaboration diagrams for static checking and test generation. In: Proceedings of third international conference on the UML, pp 385–395
5. Cavarra A, Davies J, Jeron T, Mournier L, Hartman A, Olvovsky S (2002) Using UML for automatic test generation. In: Proceedings of ISSTA 2002
6. Lugato D, Bigot C, Valot Y (2002) Validation and automatic test generation on UML models: the AGATHA approach. Electronics Notes in Theoretical Computer Science 66, No. 2
7. Offutt J, Abdurazik A (1999) Generating test cases from UML specifications. In: Proceedings of 2nd international conference on UML'99
8. Bertolino A, Gnesi S (2003) Use case-based testing of product lines. In: Proceedings of the ESEC/SIGSOFT FSE, pp 355–358
9. Briand, L.C., Labiche, Y. (2002) A UML-based approach to system testing. Software Syst Model 1(1):10–42
10. Fröhlich P, Link J (2000) Automated test case generation from dynamic models. In: Bertino E (ed) Proceedings of the ECOOP 2000, pp 472–491
11. Beer A, Mohacsi S, Stary C (1998) IDATG: an open tool for automated testing of interactive software. In: Proceedings of the COMPSAC '98—22nd international computer software and applications Conference, 19–21 August, pp 470–475
12. Menon A (2001) A comprehensive framework for testing graphical user interfaces. PhD Dissertation, University of Pittsburgh, Pittsburgh
13. Ostrand T, Anodide A, Foster H, Goradia T (1998) A visual test development environment for GUI systems. In: Proceedings of the ACM SIGSOFT international symposium on software testing and analysis (ISSTA-98), vol 23.2 of ACM Software Engineering Notes, pp 82–92
14. Menon A (2002) GUI testing: pitfalls and process. IEEE Computer. IEEE Computer Society Press, pp 87–88