Abstract

The Unified Modeling Language (UML) is the standard to specify the structure and behaviour of software systems. The created models are a constitutive part of the software specification that serves as guideline for the implementation and the test of software systems. In order to verify the functionality which is defined within the specification documents, the domain experts need to perform an acceptance test. Hence, they have to generate test cases for the acceptance test. Since domain experts usually have a low level of software engineering knowledge, the test case generation process is challenging and error-prone. In this paper we propose an approach to generate high-level acceptance test plans automatically from business processes. These processes are modeled as UML Activity Diagrams (ACD). Our method enables the application of an all-path coverage criterion to business processes for testing software systems.

1 Introduction

Testing is a crucial action throughout the complete life-cycle of software systems. The different phases of a software life-cycle involve different groups of stakeholders whose knowledge and understanding of software engineering differs heavily. This is especially true if software architects and domain experts talk about the software system. The software architect constructs the system design specification documents depending on the requirements stated by representatives of the customer’s group, namely, domain experts. The software architect has extensive knowledge about the software engineering process, while the domain expert has extensive knowledge about the functional domain but low level of knowledge or none knowledge about software engineering.

As a suitable level of abstraction used to talk about a software system, software architects employ the Unified Modeling Language (UML) which de facto is the standard of the software industry. The models created with UML usually constitute a big part of the software system’s specification documents. The specification documents are created during the initial phases of the software process to record early design decisions and stated requirements. They exist alongside the software product, serve as guidelines during the phase of implementation and are part of the documentation as well as a deliverable on completion of the product. Even though the specification documents are used to communicate the design progress of the system, the domain experts struggle to understand the contained models of the software system.

At the latest, on the time of delivery of the whole software system or at the time of delivery of fixed milestones, the domain expert needs to perform an acceptance test to verify the completeness and correctness of agreed functions. Hence, the domain expert needs to test the system, and, therefore, he needs to create the respective test cases. It is obvious that the better the test cases reflect the real usage scenario of the system, the better the quality of the result. Therefore, the test cases should be closely related to the software specification. This can be achieved if the test cases are generated from the software specification. Since a domain expert usually has a low level of software engineering knowledge, in particular, software testing knowledge, he is likely to have difficulties with the task to derive test cases from the specification documents which are predominantly UML diagrams. This can lead to an unstructured test case generation process and cause several effects. The test cases do not cover the functionality that is to be tested, and, therefore, the results are insufficient. Furthermore, the test case generation process may be not repeatable. All in all the acceptance test can become inefficient and therefore expensive.

In this paper we propose an approach which enables the domain expert to generate high-level acceptance test plans automatically from the software specification documents. The approach regards the testers viewpoint as relevant one and focusses on the workflows of modeled business pro-
cesses. We assume the business processes are modeled as UML Activity Diagrams. First, the user interaction steps within the workflow are marked with the role of the user that executes these steps. A user interaction step is any input a user states to the system or any output a user can expect from the system. In a second step the UML Activity Diagrams are transformed into an Interaction Flow Diagram (IFD) that contains only user interaction steps. Thirdly, the IFD is transformed into an Interaction Flow Graph (IFG). The IFG is directed graph that contains all paths of the IFD. In the fourth step the test cases are generated and assembled to test plans. A test plan contains all test cases to test the system. Each test case contains the proper instructions for a tester (in this case the domain expert) to execute it step by step. Furthermore, the test plan contains for each test case the inputs a tester has to provide to the system and the corresponding outputs to be expected from the system. Due to the fact that our approach makes only a few and rather less restrictive demands on the modeling of business processes, the freedom of modeling is kept.

The paper is structured as follows: Section 2 gives an overview of the related work to the proposed approach. While there are many approaches that synthesize test cases from UML diagrams, we arranged them in groups and discuss each approach within the group. Section 3 presents our contribution for generating test plans from UML Activity Diagrams (ACD), which can be considered new to existent approaches, but still in an early stage of maturity. Section 4 concludes this paper and outlines the fields of future work.

2 Related Work

In this section we summarize a number of recent research approaches to generate test cases from UML diagrams. Each approach is presented by describing the intention, classifying the abstraction level of the generated test cases, outlining the rough procedure, and discussing the relation to our approach. As there are quite a lot of contributions focusing on the test case generation from UML Diagrams, we categorised the approaches into the following groups: Activity Diagram based approaches, State Machine based approaches, and hybrid approaches.

Test Case Generation from Activity Diagrams The approaches presented here focus on the generation of test cases from UML Activity Diagrams. The presented approaches are closely related to our approach since they utilize UML Activity Diagrams as well. The approach presented by Kim et al. [12] adopts the testers’ viewpoint and therefore focuses on the inputs to the system and the outputs from the system. The test cases are derived from a constructed graph. Applying the single stimulus principle helps preventing the state explosion problem. Similar to our method, the testers’ viewpoint is regarded as relevant. Even though this approach generates valid high-level test cases, the whole process is considered to be executed manually. Our approach focuses, a strong tool support and is validated by a tool prototype. Furthermore, the approach by Kim et al. [12] involves no role model, which allows mapping operation steps to the corresponding group of users. The presented method is rather embryonic and lacks the tool support. Furthermore, we did not find any evidence of further research activity on that topic.

Chen et al. [4] propose an automatic test case generation approach that focuses on generating unit level test cases. The proposed approach does not derive test cases directly from the UML Activity Diagrams. The authors argue that the direct test case generation from Activity Diagrams requires complex algorithms due to the fact that the branches and loops within these models make the path conditions very complicated. By the approach it is intended to avoid this drawback of the state of the art of UML model-based testing. In this approach at first numerous test cases are generated randomly, and then the generated test cases are utilized to run the software. While running the software with the generated test cases, the program execution traces are elicited. The program execution traces can be used in two ways. First, they can be utilized for comparison with the UML Activity Diagram paths to determine the correctness of the implementation. Secondly, they are utilized for the derivation of a reduced set of test cases matching a defined test adequacy criterion. While our approach targets high-level system test cases, the approach of Chen et al. [4] targets low-level unit test cases. This means by the method of Chen et al. [4] it is intended to determine the correctness of algorithms and the correct interaction of classes, whereas by our approach it is intended to determine the correct support of workflows of business processes. Besides the substantial differences regarding the intention of the two approaches, the presented method by Chen et al. [4] contains some remarkable drawbacks as it restricts the modeling and implementation. For instance, the approach regards each action within the used Activity Diagrams as method call, which is a restrictive requirement regarding the modeling. In addition to that, the approach cannot differentiate between method calls to different instances of a class. Hence, it is required that each class of the system under test has only a single instance, which is a restrictive requirement regarding implementation.

Test Case Generation from State Diagrams The approaches presented here focus on the generation of test cases from UML State Diagrams. They differ from our contribution in that way that they utilize a different diagram type to synthesize test cases from. Just using another behavioral diagram for test case generation is certainly not a wide
The contribution of Fröhlich et al. [10] was the first on the particular topic of deriving test cases from use cases by transforming them to UML State Diagrams. As mentioned above, the method of Riebisch et al. [17] utilizes this transformation, too. The test case generation process comprises two general steps. First, the use cases are transformed to UML State Diagrams. Thereby, each element of the use case is mapped to the UML State Diagram. Second, test cases are derived from these diagrams. In order to generate test cases, the authors transform the test case generation problem into an AI planning problem, which can be solved with STRIPS [9], a fundamental AI planning system. Although this approach generates valid high-level system test cases, the procedure involves an extensive additional effort for the tester. For instance, to reach the traceability of the requirements throughout the whole software life-cycle the tester has to define how each part of the requirements specification impacts the downstream artifacts.

Kim et al. [13] the application of state diagrams for class testing is discussed. The approach does not address high-level tests but low-level tests of classes. The test cases are derived from UML State Diagrams satisfying a set of coverage criteria based on control and data flows.

In Gnesi et al. [11] an approach for formal test case generation from UML state charts is proposed. It is based on input-enabled transitions systems with transitions labelled by input/output pairs, which provide the semantic model for a behavioral subset of the state charts. Basically, the theoretical foundations of model-based formal test case derivation is addressed using formal conformance testing. The presented approach is rather formal and requires extensive knowledge about software engineering, especially input-enabled transition systems. Hence, the intentions of this contribution and our contribution are completely different.

Hybrid Approaches In this paragraph another two related approaches are discussed. Both concentrate on the test case generation from UML diagrams but do not fit into the above introduced categories as they integrate the information of at least two UML diagram types. In the first of the two approaches, by Nebut et al. [15], integrated UML Use Case Diagrams and UML Sequence Diagrams are used to generate test cases. In the second approach, by Bertolino et al. [2], UML Sequence Diagrams and UML State Charts are utilized to derive Test cases.

Nebut et al. [15] describe an approach to automate the generation of high-level system test scenarios for object-oriented embedded software. The authors intend to integrate formal methods with established software engineering practices to improve the feasibility of this approach for the industrial application. Their main contribution is the generation of test cases from a formalization of the requirements. The test case generation process itself is divided to two phases featuring a different level of automation. In
the first phase the requirements stated to the software system are modeled as UML Use Case Diagram. The modeled use cases are annotated with pre- and postconditions. The authors call this an requirement-by-contract approach analogous to the design-by-contract approach by Meyer [14]. The annotated use cases are transformed by a tool into a simulation model. The model consists of correct sequences of use cases - the so called test objectives - and serves as input for a simulation. The results of the simulation are used to fix detected inconsistencies of the model. The second phase of the approach is the test scenario generation. The test objectives serve as input for this generation step. The transformation of test scenarios is related to the approach of Briand and Labiche [3]. The integration of use cases and the UML Sequence Diagrams refining these use cases enables to obtain test scenarios that are closely related to the implementation. This approach concentrates on high-level system test cases as well as our approach. However, the main difference between our approach and this one is that the method presented by Nebut et al. [15] is tailored for embedded systems. Our approach focuses on information systems. A difference between information systems and embedded systems is that the state space of embedded systems is very limited compared to the state space of information systems. The simulation model that is built during the first phase is suited to clarify this issue. Applying this approach to information systems will result in a simulation model which complexity in building time and memory consumption is too high to be handled.

Bertolino et al. [2] present an approach which integrates the information of UML Sequence Diagrams and UML State Diagrams. The integrated diagram serves as input for the test case generation. One of the main goals of this approach is the industrial application. Therefore, the authors name three demands that have to be met by an approach to enable industrial application. First, the approach has to cope with a software specification that is not formally defined, but semi-formal such as UML. Second, the testing must be supported within the early phases of software development, while the software specification is neither complete nor consistent. In other words, the approach must be applicable even if the system is only partially modeled. Finally, the approach must increase the test accuracy even if the test effort is reduced. The authors identify these demands as crucial regarding the feasibility for industrial application and argue that their contribution is new in this regard with respect to existent approaches. Basically, the test case generation approach comprises two aligned procedures. First, the information of Sequence Diagrams and the State Diagrams are integrated into a more complete model. This (reasonably) complete model is checked for inconsistencies, more precisely implied scenarios, with the help of the method by Uchitel et al. [18]. Second, test cases are synthesized from this (reasonably) complete model. The test case generation utilizes the Use Interaction Test (UIT) method by Basanieri et al. [1]. The whole procedure is carried out in four steps. The approach by Bertolino et al. generates test cases that test the correct interaction of components or subsystems. In short, it generates test cases for integration tests. On the contrary, our approach generates high-level system test cases which test the system as a whole. Instead of testing the correct interaction of modules, we aim to test whether the workflows of business processes are supported correctly by the system under test. While our approach tests the system from the user’s viewpoint, the approach of Bertolino et al. tests the system under test from a rather technical viewpoint. Similarly to Bertolino et al., our method is able to generate test cases even if the system is only partially modeled. Only, the workflows which are to be tested have to meet a few requirements regarding their consistency.

3 Our Approach

In this section we will detail our approach, starting with a brief introduction to the requirements we state on activity diagrams (ACD) to be used as input for our generation method. Furthermore, we will describe the motivation for and the realization of the model transformation steps that have to be taken in order to derive test plans from business process workflows modeled as UML ACD.

3.1 Requirements on Activity Diagrams

The most basic assumption is that the business processes, that are to be supported with the software are part of the software specification and therefore are modeled with a modeling language that provides formally defined syntax and semantics. At the moment, we assume that the workflows of business processes are modeled with the Unified Modeling Language. Other modeling languages with formally defined syntax and semantics, that support the modeling of business processes are suitable, too. But the actual prototype of the tool concentrates on the UML 2.0 standard.

The Unified Modeling Language (UML) supports the modeling of processes and process related concepts in Activity Diagrams (ACD). With the use of ACDs, an abstract view onto the behavior of a system supporting the viewpoint of business processes can be modeled. For test plan generation, we suppose an abstraction level of ACD to provide rather a “functional system model” than a technical model; an example is given in Fig. 1. Furthermore, for the generation of test plans, the following (technical) requirements have to be met by ACDs:

- Functions as well as decisions, which are manually executed (by a specific user role) are explicitly marked.
3.2 Making Inputs and Outputs Explicit

The description of a business process modeled with an UML Activity Diagram (ACD) tends to be very detailed. Such diagrams usually contain process steps which are related to a user and process steps which are executed transparently by the system. User related process steps are any kind of decisions or actions that require the user to state an input to the system or to provide an output to the user. In this paper, we regard the user’s or rather the tester’s viewpoint as the relevant one and, therefore, the system as black-box. In doing so, the user is only able to state inputs to the system and observe the corresponding outputs from the system. Compiling a workflow description containing all user related process steps from a certain business process, should enable a user to execute the business process with the workflow description on hand. In the following, we call such a workflow description test plan and the user executing the steps tester.

A test plan contains proper instructions which input a user has to provide and which output is to be expected accordingly. With a test plan on hand a tester is able to verify the correct behaviour of a certain system under test (SUT) concerning the realization of the business process.

From the testers viewpoint only process steps requiring input from the tester or providing output to the tester are relevant. Hence in order to compile a test plan, these process steps have to be found, marked and filtered. Therefore, we define a special type of diagram that contains only user related process steps, the Interaction Flow Diagram (IFD).

3.3 Interaction Flow Diagram

The Interaction Flow Diagram represents the control flow of inputs and outputs a tester has to state to the system and to expect from the system. Internal actions which are transparent for the tester are suppressed and are not part of the IFD. The IFD is characterized by a drastically reduced set of node types compared to the UML Activity Diagram and serves as initial point to compile a test plan. The Interaction Flow Diagram is regarded as an in-memory model which is exclusively used as an intermediate step to generate test plans. The IFD generally serves two purposes. First, it serves as the starting point to generate test cases from, since it contains only the user interaction steps. The reduced syntax makes the processing of the contained information rather simple compared to the UML Activity Diagrams. Second, the IFD serves as an abstraction layer. Although the approach currently exclusively focusses UML
Activity Diagrams, we plan to integrate more modeling languages that are suitable to model business processes.

![Figure 2. Elements of the Interaction Flow Diagram](image)

In Fig. 2 the elements of an Interaction Flow Diagram are outlined. Basically, there are only three groups of nodes: Initial nodes, Control nodes, and User Action nodes. The group of Initial Nodes contains the Start node and the Stop node. The User Action node group is formed by the Create Action, the Manipulate Action, and the Show Action. These three types of User Actions reflect the most basic interaction concepts. The Create Action involves the creation of a particular object. The Manipulate Action involves the modification of a particular object. The Show Action involves presenting a particular object to the user. The group of Control nodes contains only the Decision node at the current state of realization. More Control nodes like Fork and Join nodes which require a more advanced handling during model transformation are left out for the sake of simplicity. All nodes contained in an Interaction Flow Diagram are vertically connected via control flow arcs.

Start and Stop nodes are equal in syntax and semantics to Start and Stop nodes in UML Activity Diagrams. User Actions are depicted as rectangles with a name and a stereotype denoting the user role. Object flows Input and Output objects are represented as horizontal arrows heading to or away from the User Action. Decisions that are depicted as ellipses are also labelled with a stereotype displaying the user role associated with this decision. A Decision node of the IFD has at least two outgoing arcs and one incoming arc. The sequence of User Actions and Decision nodes represents the interaction control flow.

Each notation element of the Interaction Flow Diagram is annotated with a user role. This role provides the information which user group is intended to execute this action or decision. This feature is to a lesser extend important to assign the correct group of testers to the actions in order to execute them, but rather to inform the tester in which context an action is to be executed. Of course a tester is able to take over more than one role during testing the system.

![Figure 3. Metamodel of IFD and IFG](image)

As a prerequisite of the transformation from a UML Activity Diagram (ACD) to an Interaction Flow Diagram (IFD), the concerned Activity Diagrams need to be flattened to atomic actions. The corresponding mapping rules are as outlined in Fig. 5 and described as follows: Each action that is associated with a user (stereotype “role”) is mapped into a User Action within the IFD. The name of the ACD action is used as name of the User Action in the IFD and the modeled user role is mapped to the associated user of a User Action. If an action in the UML Activity Diagram has any input parameter, it is transformed into Input Objects of User Actions in the IFD as well as each output parameter in UML is mapped into an Output Object in IFD. (1) illustrates the mapping rule for an UML ACD Action with input and output parameters, (2) and (3) show the mapping of UML Actions with either an input pin or an output pin. (4) All decisions that are relevant for the control flow through the UML ACD and which are associated with a user are mapped into a IFD Decision. Comparable to User Actions, the associated role is noted in the Decision element and all outgoing control flow arcs in the UML ACD are mapped to the outgoing arcs of the IFD Decision. (5) The control flow arcs between user associated UML Actions are transformed into Control Flow Arcs of the IFD between IFD User Actions. (6) If the successor of an UML Action is not associated with an user (this action is done automatically by the system), the next manually executed action becomes the successor in IFD. As an example: because “Action 2” has no assigned user role, “Action 3” becomes the successor of “Action 1” in IFD. This is due to the fact that an end user can only verify the functions provided by the user interface, which excludes internal functions.

To explain the transformation of an Activity Diagram to an Interaction Flow Diagram and further steps of our generation process, we introduce the expenses report example.
painted in Fig. 1. The UML Activity Diagram (ACD) shows the business process of creating an expenses report. A traveller creates a new report form, fills in the form, and saves this report as draft for further editing or decides to submit this form. An accountant will process the travellers report and decide to refund the travellers expenses or to send the report back to the traveller for revision. The ACD contains all necessary information and therefore is prepared for the transformation to an Interaction Flow Diagram (IFD).

### 3.4 Interaction Flow Graph

Prior to deriving test cases all paths of the Interaction Flow Diagram conforming to a selected coverage criterion are mapped to a graph. This graph is called Interaction Flow Graph (IFG) and provides the foundation for the test plan generation process. The IFG is a directed acyclic graph.
Even though the Interaction Flow Diagram can contain cycles, the Interaction Flow Graph is free of any cycle to ease the tree traversal. The absence of any cycle in the IFG is achieved by applying a modified Depth-First-Search (DFS) algorithm, which ensures that each cycle and each combination of cycles, if there are any nested cycles, are executed only once. The basic concept of this algorithm is the cycle protection. The cycle protection prevents the recursive algorithm to traverse the same cycle twice while building a path for the IFG.

The Interaction Flow Graph is built by a subset of node types of the Interaction Flow Diagram, because not all nodes are required here. Since the Interaction Flow Graph and the Interaction Flow Diagram are based on the same metamodel, the mapping of nodes from IFD to IFG is easy. Basing both IFD and IFG on the same metamodel ensures that no information that is associated with a particular node will be lost during transformation. Essentially the Control nodes are not mapped to the IFG. The Interaction Flow Graph is a directed graph and each path in this graph represents a sequence of inputs and outputs to traverse a particular business process. Hence, if the IFD contains a cycle, the nodes associated with this cycle will appear more than once in the IFG.

For the business expenses example we select the all-paths-coverage criterion. The example contains two cycles of which the first cycle is a nested cycle. The first cycle starts with the transition from Decision D1 to “Fill Form” and the second one leads from Decision D2 to “Fill Form”. Due to the fact that the second cycle includes the start of the first cycle, the first cycle is nested in the second cycle. Nested cycles lead to the effect that each combination of execution of the cycles results in a distinct path. Usually the application of the all-paths-coverage criterion to a process with nested cycles is complex since the traversing algorithm risks to run into one of the cycles.

As mentioned above, we employ a modified Depth-First-Search algorithm to walk through the Interaction Flow Diagram. Essentially, our algorithm inspects the nodes and their successors of the IFD and marks each visited node and cycle in order to avoid running into a loop. In doing so, each node that is a subtype of IFGNode is put into the graph. Fig. 7 shows the example Interaction Flow Graph derived from the Interaction Flow Diagram illustrated in Fig. 5.

The Interaction Flow Graph now contains all paths derived from the Interaction Flow Diagram. These paths are extracted and analyzed. To extract these paths, a feasible algorithm must be applied. Since the graph contains no cycles by definition, there are many feasible algorithms to extract the paths from the graph, e.g., Breadth First Search, or Depth First Search.

Depending on the number of Decision nodes and nested cycles contained in the Interaction Flow Diagram, the num-

```
create a start node in IFG;
for each (t in transitions with start node as source) {
    put t into Q;
}
while (Q is not empty) {
    take transition t from Q;
    get target node n from t to IFG;
    if (n is already a node in IFG) {
        mark t as "cycle-transition";
    }
    add n to IFG;
    for each (t in transitions with n as source) {
        put t into Q;
    }
}
for each (t1 in transitions in IFG marked as "cycle-transition"){
    get target node n from t1;
    for each (t2 in transitions with n as target) {
        if (t2 is not marked as "cycle-transition") {
            add t2 to IFG;
        }
    }
}
```

Figure 6. Applied Modified Depth-First-Search Algorithm

Figure 7. Interaction Flow Graph
ber of paths extracted to the IFG can be quite huge. As Fig. 7 shows, our algorithm found five paths for the expenses report example. We call each path a test scenario and each single testing step within a scenario a test case.

### 3.5 Construct Test Plans

The final result of the test plan generation is a human-readable representation of a test plan. A domain expert is now able to process a (technical) structured acceptance test without having explicit knowledge about testing software-systems. The generated test plan guides the domain expert who has to check whether the system behaves as expected or not through the system. In Fig. 8 and Fig. 9, the tester is provided with instructions what is to be done. The output shows five test scenarios with thirty-five test cases. Each test case is presented with the information, which user role executes this process step, what is to be done, and what is to be expected.

In the first test scenario, the tester with the role “Traveller” has to “create expenses report” and expect “fill form” to be shown. In the next step, the tester with the role “Traveller” has to “fill form” and decide to “send to accountant”. Before the tester with the role “Traveller” can “confirm” the calculated refund, the tester with the role “Accountant” has to “check validity” of the report and decide that the “report is valid”. Afterwards the tester with the role “Accountant” has to “calculate refund” and expect the “transfer refund”.

This simple example shows that the generated test plan provides simple instructions for a tester to execute a business process with the system under test. The tester is guided through the testing process and can ensure that the selected coverage criterion is applied. While the tester is guided through each path of the business process, it can be verified if the system behaves as specified or not.

### 4 Conclusion

In this paper we presented an approach to generate high-level acceptance test cases from business processes, that are modeled as UML Activity Diagrams. We regard the system under test as a black box, and, therefore, the tester’s viewpoint as relevant. This is accomplished by concentrating on inputs a tester makes to the system and outputs a tester can observe from the system. To process such inputs and outputs, we apply several model transformation steps. A first step transforms the business process to an intermediate model called Interaction Flow Diagram (IFD). The next step creates a graph that contains all paths from the intermediate model. From this graph we derive the test cases that are assembled in test plans. The test plans structure and guide the testing activities and therefore support the domain expert during the acceptance test. Since our approach analyses modeled business processes, only the information that is modeled can be included in the test plans. Therefore, the modeled business processes should provide rather detailed information. Furthermore, the approach focusses the test of functional requirements. Even though we focussed this paper on acceptance tests, the approach is also fitted to support the system test. However, one advantage over other approaches is that it enables domain experts that have none or less software engineering knowledge to generate test cases for the acceptance test from the software specification on their own.

To prove the proposed method, we implemented a prototype that is able to import a business process modeled as UML Activity Diagram, apply all transformation steps on

---

### Figure 8. Example of a generated test scenario summarized within a table.

<table>
<thead>
<tr>
<th>Test Scenario 1</th>
<th>Role</th>
<th>Testcase ID</th>
<th>Input</th>
<th>Output</th>
<th>Summary</th>
</tr>
</thead>
<tbody>
<tr>
<td>Traveller</td>
<td>TC2</td>
<td>Fill Form</td>
<td>Fill Form; provide input Report; validate output Report; Send to Accountant;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traveller</td>
<td>TC3</td>
<td>Fill Form</td>
<td>Fill Form; provide input Report; validate output Report; Send to Accountant;</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Accountant</td>
<td>TC4</td>
<td>Calculate Refund</td>
<td>Transfer Refund; provide input Report; validate output Report; expect Transfer Refund</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Traveller</td>
<td>TC5</td>
<td>Confirm</td>
<td>Confirm;</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Figure 9. Example of a generated test case with detailed information.

#### 1.1.3 Testcase 3

<table>
<thead>
<tr>
<th>Role: Traveller</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input: Report</td>
</tr>
<tr>
<td>State: Invalid</td>
</tr>
<tr>
<td>A report is invalid if e.g. not all required documents of the report are received.</td>
</tr>
<tr>
<td>Output: Report</td>
</tr>
<tr>
<td>State: Filled</td>
</tr>
<tr>
<td>All required fields of the report are filled.</td>
</tr>
<tr>
<td>Procedure: Manipulate</td>
</tr>
<tr>
<td>Action: &lt;Manipulate Action&gt; Fill Form</td>
</tr>
<tr>
<td>Enter itinerary, specify bills and receipts</td>
</tr>
<tr>
<td>Validate:</td>
</tr>
<tr>
<td>Decision: Send to Accountant; Save as Draft</td>
</tr>
<tr>
<td>Save as draft if not all documents are received.</td>
</tr>
<tr>
<td>Assign to accountant if report is complete.</td>
</tr>
<tr>
<td>Expect:</td>
</tr>
</tbody>
</table>

---
that diagram and assemble the respective test plans. Fig. 8 and Fig. 9 are showing two details of a generated test plan which was generated by the implemented prototype. Fig. 8 shows the table that summarizes all test cases of a certain test scenario. Fig. 9 shows a sample of the detailed description of the test case. This detailed description provides the structured information how the tester has to interact with the system. It details which input the tester has to provide, which output is to be expected and what actions the tester has to take. The prototype is implemented as an Eclipse plugin [7] using EMF [8] - the Eclipse Modeling Framework. The prototype is able to import UML Activity Diagrams that are exported as XMI [16] - the XML Metadata Interchange format file. XMI is the most popular format for exchanging UML models.

Although our approach is still in an early stage of maturity, the tool prototype shows the general feasibility of the proposed method. Furthermore, we believe that the method is valuable contribution to the improvement of test efficiency. Our long-term objective is to advance this procedure to a method that specifies the generation of test cases that can be executed automatically within a test harness. We are addressing the issue of test automation within a research project that focusses on testing web-based information systems. Therefore, we have to address several concerns like, the correct handling of fork and join nodes during model transformation, the integration of object states and the user actions, and the selection of valid test data. A further field for future work is the integration of other modeling languages that are suitable to model business processes.

References