Abstract:
Use cases defined as a tabular text are a widely used technique in the early phases of system development. However, there is a lack of approaches to derive test objectives and test cases from these use cases. This paper introduces a set of processes that allow the generation of test objectives as use case scenarios and operational variables. Although use cases are expressed in a narrative style, the process has been automated with two supporting tools.

1. INTRODUCTION

Nowadays, use cases are a widely used technique to define the functional requirements of software systems. For this reason, they are also a main testing artefact; the test phase must assure that the final software system satisfies its use cases.

There are two common techniques for extracting test cases from use cases: scenario analysis and test data analysis. Scenario analysis technique is based on the identification of scenarios from the use case. Every scenario is a test case candidate. Test data analysis is based on the category-partition method [15]. First, a set of categories, also called operational variables in [2] and also in this paper, are identified from the use case. Operational variables denote every element that may be different between two scenarios from the same use case. Then, the domain of every variable is divided into partitions. Finally, every combination of partitions is a test case candidate. Several papers (see section 4) use only one of the two techniques. However, there is a trace dependency between use case scenarios and combinations of partitions. On one hand, a use case scenario may only be performed using some concrete combinations of partitions. On the other hand, a combination of partitions executes behaviour described as one use case scenario. There is, also, a lack of systematic approaches exposing how to automate and to combine both techniques. This paper introduces a fully automated approach to generate test objectives (scenarios plus combinations of partitions) from use cases.
In general, a test objective is something that must be tested [16]. In other words, test cases are designed to implement the defined test objectives. In the scope of this paper, the test objectives that test the implementation of use cases are the use case scenarios and their operational variables. Test cases might be generated to exercise each use case scenario and each combination of operational variables to make sure that the answers of the system match up with the definition of the use cases. We use the "test objective" concept instead of "test case" concept because artefact generated in next sections do not include the expected results, thus they are not test cases.

The approach described in this paper is divided into three processes. The first process describes the translation of the use case definition into a UML activity diagram (sections 2.1 and 2.2). The second process describes the generation of use case scenarios using the activity diagram (section 2.3). And the third process describes how to identify operational variables and domains from the decision nodes of the activity diagram and how to generate valid combinations of values for the operational variables (section 3). Then, section 4 summarizes other related approaches for testing use cases. Finally, section 5 presents conclusions and ongoing works.

2. GENERATION OF USE CASE SCENARIOS

2.1. Definition of use cases

Use cases are a powerful and flexible technique for requirement elicitation. However, it is possible to define use cases in several different but complementary ways and with several levels of details. UML notation offers the Use Case diagrams to define the use cases of a software system. However, the use case metaclass only defines the name of the use case and its relations with actors and other use cases, but it does not define its behaviour. Several author, like Cockburn [4], Dustin [6] or Escalona [7], have proposed textual templates to define the behaviour of use cases in detail. Use case templates express the behaviour using steps, which are complete sentences with subject and predicate and they are written in a non-formal language. The use case steps and other elements are structured through tabular forms. This paper uses the approach defined by Escalona (called Navigational Development Techniques – NDT).

First, the use case is codified as a XML document, using the same structure than the template, to improve automation, as seen in the example of table 1. The XML structure may be easily extended to include other elements of a use case, as exposed in the mentioned references. DTD available in www.lsi.us.es/~javierj
The alternative steps define the behaviour that may be realised as an optional alternative to a step of the main sequence. The erroneous steps define the behaviour that may be used if a step from the main sequence meets an error and is unable to exercise its behaviour.

The granularity, or level of detail, has to be considered too [17]. Cockburn [4] defines three main kinds of levels: system scope, goal specificity and interaction detail. The most adequate level for this approach is the goal specificity level, because it describes a short interaction (seconds or few minutes) among the system and one or a little number of actors to achieve a goal that is easy to verify in the system under test. The use case in table 1 is defined at goal specificity level. The size of a use case, this means the number of steps, is not relevant.

Following sections show a practical case study, which illustrate each process. The system under test is a web application for the storing and querying of URL links (provided by www.codecharge.com). Use case, in table 1, describes a search operation of this web application but the approach may also be applied over other types of system, like desktop applications.

2.2. From use cases to activity diagrams

The natural language is often too ambiguous and generic to be automatically processed. An intermediate approach is the use of language patterns. One pattern offers a static phrase with some variable elements than a user or engineers may change. An example of pattern, for the definition of an alternative behaviour, is showed in table 2.
Table 2. Transformation of a step into an activity diagram.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>If [CONDITION] then [ACTION] and this use case ends.</td>
<td></td>
</tr>
</tbody>
</table>

Pattern in table 2 is used in erroneous steps, 4.2 and 4.3, in the example use case (table 1). The next task is to translate the behaviour (main, alternative and erroneous steps) of a use case into a model. A UML activity diagram model has been chosen to define the behaviour of a use case. It clearly expresses the interactions between the actors and the system, it indicates if an action is performed by the system or by an external action and it includes different execution flows. Activity diagrams do not need to expose information about the implementation of the system or the implementation of its external interfaces. An example of use case is showed in table 1 and its activity diagram is showed in figure 1.

Each use case definition is automatically translated into one activity diagram. To perform this process, a set of pattern language and translation algorithms is needed. The pattern language allows the automatically extraction of the relevant information from the steps of a use case so that the algorithms can build the activity diagram from that information. An example of pattern from the language and its translation into a fragment of an activity diagram is shown in table 2.

The algorithms used to extract information from a use case and to generate an activity diagram may be consulted in previous papers, like [9]. They have been implemented in a open-source software tool called ObjectGen (available in www.lsi.us.es/~javierj/).

The activity diagram indicates if an action is performed by the system under test or by an actor using stereotypes.

2.3. Derivation of use case scenarios

Use case scenarios are derived from the activity diagram generated in section 2.2. Each path of the activity diagram is a possible use case scenario. Classical graph-coverage techniques may be used to derive use case scenarios, like Round-strip pattern [2] or all-nodes and all-transitions coverage criteria [10]. The ObjectGen tool implements the all-nodes, all-transitions, and round-trip (for zero and one lopping traverse) criteria to select scenarios.
Table 3. Paths and a use case scenario.

<table>
<thead>
<tr>
<th>Use case: Search link by description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenarios: 5</td>
</tr>
<tr>
<td>S 1: 1, 2, D01, End.</td>
</tr>
<tr>
<td>S 2: 1, 2, D01, 3, D02, 4.1, 5, End.</td>
</tr>
<tr>
<td>S 3: 1, 2, D01, 3, D02, 4, D03, 4.2, End.</td>
</tr>
<tr>
<td>S 4: 1, 2, D01, 3, D02, 4, D03, D04, 4.3, End.</td>
</tr>
<tr>
<td>S 5: 1, 2, D01, 3, D02, 4, D03, D04, 5, End.</td>
</tr>
</tbody>
</table>

Use case scenario 1:
1: The visitor asks the system searching for links by description.
2: The system asks for the description.
3: The visitor cancels the search then the use case ends.

Table 3 shows an example of the paths and use case scenarios that have been obtained after applying the round-trip criterion in the activity diagram.
3. GENERATION OF COMBINATIONS OF PARTITIONS

3.1. Definition of operational variables and partitions

Every decision node, in the activity diagram, evaluates a logical predicate to choose an output transition. Therefore, an operational variable is assigned to each decision node and logical predicate. The domain of the operational variable is divided into as many partitions as output transitions. Two partitions have been identified for every operational variable in the example activity diagram (figure 1). The first partition contains the values that make the predicate true. The second partition contains the values that make the predicate false. Another open-source tool, called ValueGen (also available in www.lsi.us.es/~javierj), has generated the list of variables and partitions as a XML structure (table 4).

Table 4. List of operational variables and sub-domains

```xml
<?xml version='1.0' encoding='UTF-8' ?>
<variables>
  <variable id="V_D01">
    <partition id="P_1">
      <condition> Cancel the search </condition>
    </partition>
    <partition id="P_2">
      <condition> Not(cancel the search) </condition>
    </partition>
  </variable>
  <variable id="V_D02">
    <partition id="P_1">
      <condition> The visitor introduces an empty description. </condition>
    </partition>
    <partition id="P_2">
      <condition> Not(the visitor introduces an empty description) </condition>
    </partition>
  </variable>
  <variable id="V_D03">
    <partition id="P_1">
      <condition> The system finds any error performing the search, </condition>
    </partition>
    <partition id="P_2">
      <condition> Not(the system finds any error performing the search) </condition>
    </partition>
  </variable>
  <variable id="V_D04">
    <partition id="P_1">
      <condition> The result is empty, </condition>
    </partition>
    <partition id="P_2">
      <condition> Not(the result is empty,) </condition>
    </partition>
  </variable>
</variables>
```
The list of operational variables may be completed adding new variables by hand. In the same way, new partitions may be added or some partitions may be divided again.

3.2. Construction of the dependency matrix and generation of constraints

Frequently, the value of a set of operational variables is conditioned by the values of other operational variables. For example, the activity diagram (figure 1) shows clearly that, if the variable V_D01 associated to the decision node D01 takes a value from the partition "P_1" (this means: the visitor cancels the search operation), then the use case ends, variables V_D02, V_D03 and V_D04 do not need any values and no more combinations have to be calculated for these three variables. Therefore, if all the possible combinations among the values of the operational variables are calculated now, then 16 combinations are obtained. However, not all combinations are valid ones.

So, the next step is the generation of constraints for avoiding wrong combinations. Three artefacts are needed for generating constraints automatically: the list of use case scenarios (generated in section 2.3), the list of variables (generated in section 3.1) and the dependency matrix of those variables. Next paragraphs show how to generate the dependency matrix.

A dependency matrix stores the variables that depend on the values of other variables. A dependency matrix is a binary or boolean matrix with as many rows and columns as operational variables generated in the previous section. Table 5 presents an example of dependency matrix generated by the ValueGen tool from the activity diagram of figure 1 and the variable list of table 4.

<table>
<thead>
<tr>
<th>V_D01</th>
<th>V_D02</th>
<th>V_D03</th>
<th>V_D04</th>
</tr>
</thead>
<tbody>
<tr>
<td>V_D01</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>V_D02</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>V_D03</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>V_D04</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

For example, the value of variable V_D02 depends of the values of variable V_D01. Thus, column V_D01 shows a “1” in the intersection with row V_D01. As showed in the dependency matrix of table 5, the variables do not depended of themselves.

It is possible to generate a set of constrains to avoid the sequence of invalid combinations with the dependency matrix and the elements obtained in previous sections. If a use case scenario contains a variable V which takes a value from its partition P, the variables W and X depends on V and variables W and X do not appear in the use case scenario, then, a constraint has been identified. This constraint says: if variable V takes a value from its partition P, no values are needed for variables W and X.
The ValueGen tool also generates an executable script (in BeanShell language) to calculate all the obtained combinations taking into account the constraints. Three variables (V_D01, V_D02 and V_D03) have dependent variables, thus, no more than three constraints may be identified. The results of the execution of this script are shown in figure 2. A "*" character denotes that a variable does not take any value.

The number of combinations has been reduced from 16 to 5, less than 33%. There are five scenarios and five combinations, thus each combination of partitions are associated with one scenario. For example, scenario S 5 (table 3) describes de successful scenario; its associated combination is also the number 5 (figure 2).

4. RELATED WORKS

There are several papers and approaches about the testing of use cases defined in a textual tabular notation. An extensive list of references may be found in [5] and [8]. Next paragraphs describe some of the most relevant approaches.

Binder [2] describes the Extend Use Case Test pattern, based on the Category-Partition method. This pattern is focused on the identification of operational variables and builds up a decision table with all the combinations between values and the expected results for each set of values. The main ideas of section 3 have been extracted from this pattern; however, it is less formal than our approach and does not support any automation nor generate test scripts.

TDE/UML approach, [16], expresses a use case as a UML activity diagram and uses the Category-Partition method [15] to generate test cases. The diagram is annotated with variables, categories, partitions and conditions. A proprietary test tool calculates all the possible combinations between the paths and categories that commit all the conditions. However, the approach does not indicate if the activity diagram may be generated automatically from the use cases, nor the format in which the use cases must be defined. TDE/UML might generate executable test scripts, but do not generate any expected results and validation actions.

TOTEM [12], Requirement-based Contract [14] and the CowSuite [1] approaches expressed a use case as an UML sequence diagram. The sequences of
messages are expressed as regular expressions and are combined between them to generate test cases. We found some problems using sequence diagrams. It is very difficult to express alternative or erroneous sequences in the same diagram. Information about architecture and internal implementation, like classes and messages are also needed, so it cannot be applied in the early phases of the development.

RETNA approach [3] uses a paragraph of non-format text to define the use case and it applies language processing techniques for extracting information, building state machines and generating test cases. This approach introduces strong restrictions in use cases, and it needs a wide amount of tools and techniques, mainly of language processing, which makes it difficult to apply.

Other approaches work directly with natural language, like references [11] and [13]. All of them propose a simple combinatorial explosion among all scenarios in a use case. These approaches are quite simple and omit many important aspects, like coverage, test values, expected results or test implementation.

5. CONCLUSIONS

The approach introduced in this paper may be applied in the early stages of the development process, when use cases are being defined, because it does not need a set of complete use cases. The full automatism of the process and the availability to support tools allow the easy re-generation of test objectives when use cases change.

There are many variants of use case specification style and formality [17]. This approach only uses main, alternative and erroneous steps which are present in all styles, so, it may be easily used without having to change the use case style and format of an organisation. Another important benefit of this approach is that it allows the managing of the traceability of the test cases from a use case through their test objectives. Thanks to a set of test objectives generated from a use case and a set of test cases codified to satisfy the test objectives, it is possible to know the coverage of the use cases in test cases. In the scope of this paper, a 100% coverage is obtained when the test cases exercise all the use case scenarios and all the combinations for operational variables of the use case. This approach also emphasizes the definition of good and complete use cases in the early stages of the development and allows an additional return of the time inversion in the use cases.

Tools described in this paper are only proof of concepts, but they are not mature ones. So, it is not possible to study the performance of the approaches described in this paper. An ongoing work is the improvement of these tools and their integration with the use case supporting tool for NDT.
6. ACKNOWLEDGES

This work is supported by the Ministry of Science and Education (Spain) under the National Program for Researching, Development and Innovation, project QSimTec (TIN2007-67843-C06-03) and REPRIS (TIN2005-24792-E).

7. REFERENCES


