Generating User Acceptance Test Plans from Test Cases

Karl R. P. H. Leung
Department of Information & Communications Technology,
Hong Kong Institute of Vocational Education,
Hong Kong
kleung@computer.org

W. L. Yeung
Department of Computing
and Decision Sciences
Lingnan University, Hong Kong
wlyeung@ln.edu.hk

Abstract

There are numerous studies on generating test cases from classifications. These test cases consists of related classes from different classifications. On the other hand, a User Acceptance Test Plan (UAT) consist of test plans. Each of these test plans is a scenario of sequence of input together with the corresponding output in association with the concerned processes. A test case can conform with one or more of these scenario. In real life system testing, the number of test plan in UAT is huge. A method which generates test plans from test cases would make the test cases generation study more comprehensive. In this paper, we report our study on the generation of test plans from test cases. Test plans are generated from test cases with reference to the abstract system architecture of the system. Furthermore, our method is implementable, i.e. the test plan generation can be automated. Consequently, the amount of time required to formulate a UAT can be reduced drastically.

1 Introduction

User Acceptance Test (UAT) is regarded as the most important test in software development as once the UAT is passed, the system will be accepted by the users. Users, assisted by software developers, go through the *test plans*, in order to validate that the conformity of the system developed with the system requirement so as to convenience users to accept the system. In principle, the test plans should cover all important scenarios. Developing UAT test plans is a tedious task. Automatic test plan generation would be very useful to help to reduce the amount of time for developing UAT.

Specification based test case generation is one of the general approaches. Specification based test case generation is a kind of black box testing because the software is viewed as a black box that transforms inputs into outputs based on the specification of what the software is supposed to do. This approach considers the external views of the software and hence is a testing of the product, but not the design decisions or program code. Classification Tree (CT) method [9, 8] is one of the well studied specification based test case generation methods. Chen *et al* have studied methods for building CT from informal specifications [10, 12, 11, 13]. However, considerable manual effort is required in selecting legitimate test cases from all the test cases generated from the tree. Hence these works cannot lead to automation of test case generation nor test case generation on demand.

Instead of using classification tree, we have studied graph with logical operators [3] and class vector [6, 5, 7] for modeling the input space. These two approaches can generate all the test cases without manual decision. The notions of class and classification are formalized as well. Hence these method can be a candidate for automatic test case generation. However, these methods generate all the test cases as once. Although algorithms can be developed to extract the test cases that are required, the processing is clumsy. We found that hypergraph [1] is a better candidate than graph and vector for test case generation on demand. It is because there are many studies of using hypergraph for database browsing, querying and updating [14]. Hence, when the input space is captured by hypergraph, the model can be transformed into a database model. Then test cases can be generated from the database model with reference to the hypergraph model [4]. For convenience, we call this family of approaches, i.e. the classification tree, class graph, class vector and hypergraph, the classification approach.

UAT test plans state the sequence of inputs in association with the desired processes together with the expected outputs from these processes. The test cases generated from classification approach states the combination of classes in the tests. Test cases do not show whether the classes are inputs or outputs nor the processes. Consequently, there is a gap between UAT test plan and test cases generated from classification approach. Hence the test cases generated from classification approach cannot be used directly as test plans in UAT.

In this paper, we report our study of generating UAT test plans from test cases. We illustrate this approach by using a case study, the example is about Reward Points Handling of Credit Card which was used in [12] albeit with a number of minor changes as well as including a system architecture.

The rest of the paper is organized as follows. Section 2 describes the credit card reward system. Section 3 gives the definitions of major terms being used in this paper. Section 4 discusses the algorithm for generating test plans. Finally, we discuss our approach in Section 5 and draw a conclusion in Section 6.

2 A Case Study

The ABC Bank issues credit cards to approved customers. These customers will earn certain ‘reward points’ for every dollar charged to their cards. The number of reward points earned by each customer then determines the type of benefit (e.g. free air-line tickets, free shopping vouchers, etc) which the customer is entitled to. For each purchase, the program shall accept the transaction details together with the various information of the card. Thereafter, validation of these details is performed in order to determine whether this purchase is approved. If the purchase is approved, the program will calculate the number of reward points.

There are three classes of credit cards, namely platinum, gold and classic. The credit limit for classic cards is \$1000. The credit limit for gold cards and some platinum cards is \$4000. The credit limit for the rest of the platinum cards is \$8000.

Card holders can redeem air-tickets or products with their reward points. Air-tickets can only be redeemed from airlines CX or DG. All classes of tickets, including first, business or economy class can be redeemed from CX. Only first or business class air-tickets can be redeemed from DG. Products can be redeemed from suppliers RedA or TDC.

The minimum transaction amount should be greater than \$0. The cumulative balance is the total purchase amount of all outstanding transactions. The amount of cumulative balance should be greater than \$0 but no greater than the approved credit limit of the card.

We list the classifications and classes of the case study in Table 1.

classification	associated classes
card class	Platinum, Gold, Classic
credit limit	1000 (L1), 4000(L2), 8000 (L3)
redeem type	air-ticket, product
airline	CX, DG
air-ticket type	first, business, economy
supplier	RedA, TDC
purchase balance	$0 < A \leq 1000$ (A1), $1000 < A \leq 4000$ (A2), $4000 < A \leq 8000$ (A3), $A > 8000$ (A4)
cumulative balance	$0 < B \leq 1000$ (C1) , $1000 < B \leq 4000$ (C2), $4000 < B \leq 8000$ (C3)
approval result	accept, reject

Table 1. Classifications and its associated classes

Four processes are introduced to this credit card example. First the process *creditAuthorization(Input(card class, credit limit, purchase balance, cumulative balance), Output(approval result))* validate the credit condition of the card with the inputs in the $Input()$ and return with $Output()$, i.e. $approval result$ of either $accept$ or $reject$. The processes *redeem(Input(redeem type), Output(airline))* and *redeem(Input(redeem type), Output(supplier))* are processes for customers to choose the type of redemption. The process *chooseTicketClass(Input(airline, air-ticket type), Output(approval result))* let customers choosing the type of tickets.

3 Definitions

A UAT is a set of test plans where each test plan is a sequence of processes together with some controlled input and associated output. The input and output are tuple of classes. It should be noticed that the classes in input and output need not be distinct.

$$UAT = \{testplan\}$$

$$\begin{aligned} testplan = \quad & process_1(input_1,\ output_1); \\ & ..\ ; \\ & process_n(input_n,\ output_n) \end{aligned}$$

A test case generated from classification approach is a set of classes $\{c_1,\ ..\ ,c_m\}$. From our credit card example, the test cases that can be generated are shown in Figure 1.

A system architecture can be denoted as a tuple $< Process,\ ProcessDependency >$ where *Process* is the set of processes in the form $process(InputClassification, OutputClassification)$ of the system. *InputClassification* and *OutputClassification* are sets of classifications being taken as input and output respectively by the process. In the credit card example, the set *Process* contains the processes *creditAuthorization*, the two *redeem* processes, and *chooseTicketClass*. *ProcessDependency* is the set of ordered pair of processes $< process_1(input_1, output_1), process_2(input_2, output_2) >$ where $input_2 \subseteq output_1$. *ProcessDependency* contains the dependency relations <*redeem*({*redeem type*}, {*airline*}), *chooseTicketClass*({*airline, air-ticket type*}, {*approval result*})>.

Furthermore, let *StartProcess* and *EndProcess* be sets of processes where processes in *StartProcess* do not depend on any other process and processes in *EndProcess* have no dependent process. Then in the credit card example, *StartProcess* contains processes *creditAuthorization* and the two *redeem* processes while *EndProcess* contains the process *creditAuthorization* and *chooseTicketClass*. In this example, it can be seen that a process can be an element of both *StartProcess* and *EndProcess*.

{platinum, L3, A3, C3, accept},
{platinum, L2, A2, C2, accept},
{Gold, L2, A2, C2, accept},
{classic, L1, A1, C1, accept}
{platinum, L3, A3, C3, reject},
{platinum, L2, A2, C2, reject},
{Gold, L2, A2, C2, reject},
{classic, L1, A1, C1, reject}
{air-ticket, CX, first, accept},
{air-ticket, CX, business, accept},
{air-ticket, CX, economy, accept},
{air-ticket, DG, first, accept}),
{air-ticket, DG, business, accept}),
{product, ReDA, accept},
{product, TDC, accept},
{air-ticket, CX, first, reject},
{air-ticket, CX, business, reject},
{air-ticket, CX, economy, reject},
{air-ticket, DG, first, reject}),
{air-ticket, DG, business, reject}),
{product, ReDA, reject},
{product, TDC, reject}

Figure 1. Test cases

4 Generating Test Plan from Test Case

The objective of generating a test plan from a test case is deriving a test plan from a given test case with reference to the system architecture. The algorithm in Figure 2 is developed for such purpose.

4.1 Discussion of algorithm

The algorithm in Figure 2 generates a test plan from a given test case *testcase* obtained from classification approach. Line 1 gets a process from *StartProcess*. The function *processClass* in line 2 derives classes from the input and output classifications of process p. But the all derived classes of p must be elements of *testcase* as stated in line 3. Then if the union of all the input and output classes derived from p is equal to *testcase* and p is in *EndProcess*, i.e. all the classes in *testcase* have been included in either the input and output of the process p, and p will not lead to further processing, then $p(I_c,\ O_c)$ is one of the test plan of the *testcase* (Line 4 and 5).

```
1. get a process $p(I,O)$ from $StartProcess$
2.     where $processClass(p(I,\ O)) = p_c(I_c,\ O_c) \wedge$
3.             $I_c \cup O_c \subseteq testcase$
4. if $(I_c \cup O_c = testcase) \wedge (p \in EndProcess)$ then
5.     $testplan := p(I_c,\ O_c)$
6. else
7.   $\{TmpTestPlan := p$
8.   repeat
9.   get a dependent process $p_d(I_{p_d},\ O_{p_d})$ of
         $tail(TmpTestPlan)$
10.      where $(processClass(p_d(I_{p_d},\ O_{p_d})) =$
              $p_{d_c}(I_{p_{d_c}},\ O_{p_{d_c}}))\wedge$
11.             $(I_{p_{d_c}} \cup O_{p_{d_c}}) \subseteq testcase$
12.  $TmpTestPlan := TmpTestPlan; p_d(I_{p_{d_c}}),\ O_{p_{d_c}}$
13.  until $((\cup Input_{TmpTestPlan})\cup$
              $(\cup Output_{TmpTestPlan}) = testcase)\wedge$
14.         $(p_d \in EndProcess)$
15.  $testplan := TmpTestPlan$
16.  }
```

Figure 2. Algorithm for generating test plan

If the condition in line 4 does not satisfy, this implies that there are still some classes of the *testcase* have not been included in the test plan or the test plan still needs to incorporate some more processes. The steps starting from line 7 is dealing with this situation. First, in line 7, the process p is assigned to the $TmpTestPlan$. $TmpTestPlan$ is the variable for storing the working test plan. Line 8 starts the iteration of searching for processes to complete the test plan. The first step in the iteration (line 9) is to get a dependent process p_d of the last process of $TmpTestPlan$. The function $tail()$ returns the last process of a sequence of processes. Similar to the condition in line 2 and 3, line 10 and 11 is specifying the condition that all the derived classes of p_d must be elements of $testcase$. This process is appended to $TmpTestPlan$ in line 12. The notation "$a;b$" means process a is followed by process b. The iteration terminate when the following two conditions (line 13 and 14) holds. First, the union of all the input and output classes in all the processes in the $TmpTestPlan$ is equal to the $testcase$. Second, the last process of $TmpTestPlan$ is in $EndProcess$, i.e. it will not lead to the processing of another process. Then the sequence of processes together with the input and output is a test plan (line 15).

4.2 Necessary and Sufficient Conditions

It should be noticed that the conditions in line 2 and 3 as well as line 13 and 14 are necessary and sufficient conditions to determine the termination of the iteration. It is because all the classes in the test case should have been included in the input and output of the test plan. On the other hand, the test plan should not include classes that are not included in the test case, otherwise, the test plan is testing a scenario that does not match the test case. Furthermore, the last process in the test plan is not leading to the processing of some other processes. Otherwise, some other classes may be involved and hence the test plan is incomplete.

The condition in line 10 and 11 are necessary condition for selecting a process. It is because if there are classes in the input or output of the process that are not elements of test case, these classes are beyond the scope of the testing purpose of the test case and hence the process is not a target process.

4.3 Illustration

Let us illustrate the algorithm with examples. First, let the test case be $\{platinum, L3, A3, C3, accept\}$. In line 2, the process *creditAuthorization* is obtained. The classes *platinum, L3, A3, C3* and *accept* are in the input and output sets respectively. The union of the input and output sets is a subset of the test case. In line 4, not only the input and output sets is equal to the test case, but also the process *creditAuthorization* is in *EndProcess* as well. Hence the process *creditAuthorization* is a test plan and output from the $testplan$. It should be noticed that values of $L3, A3$ and $C3$ have to be determined in order to complete the test plan. These values can be obtained some many methods such as random testing or partition testing[2].

Let us look at another example $\{air\text{-}ticket, CX, first\}$. In line 1, the process *redeem(*$\{$*product type*$\}$*,* $\{$*airline*$\}$*)* is chosen because the classes *air-ticket* and *CX* are contained in the input and output sets respectively. However, the union of the input and output sets does not equal to the test case, hence the condition in line 4 is not satisfied. Then the process *redeem*($\{$*product type*$\}$, $\{$*airline*$\}$) is assigned to the working test plan $TmpTestPlan$ in line 7. It should be noticed that the process $redeem(\{producttype\},\{supplier\})$ is not selected because the classes of *product* are *RedA* and TDC which do not match the class CX. The process *redeem* is the last process in the test plan $TmpTestPlan$ at this moment.

The process *chooseTicketClass* is chosen in line 9 because the process is a dependent of process *redeem* and the union of the input and output set $\{CX, first, accept\}$ is a subset of the test case, i.e. satisfied the conditions in line 9 to 11. Then *chooseTicketClass* is appended to $TmpTestPlan$ in line 12. At this moment, the union of all input sets in the $TmpTestPlan$ is $\{$ *air-ticket*, $CX, first$ $\}$ and the union of all the output sets in $TmpTestPlan$ is $\{CX, first, accept\}$. The union of these two unions is $\{$*air-ticket*$, CX, first, accept\}$ which is equal to the test case. Furthermore, the process *chooseTicketClass* is in the set *EndProcess*. Consequently, the conditions in line 13 and 14 are satisfied, the iteration terminates and the $TmpTestPlan$ is output from $testplan$ in line 15. Otherwise, the process will be repeated from line 8 again until the conditions in line 13 and 14 are satisfied.

The test plan obtained with test case $\{$ *air-ticket, CX, first, accept* $\}$ is *redeem*($\{$*product type*$\}$, $\{$*airline*$\}$); *chooseTicketClass*($\{$*airline, air-ticket type*$\}$, $\{$*approval result*$\}$) . When testing with this test plan, users first input *air-ticket* to the process *redeem*. The process will output CX, DG for the users to choose and the operation goes to the process *chooseTicketClass*. Users then select CX and $first$ in the process *chooseTicketClass*. After some internal processing, the process *chooseTicketClass* return with output *accept* and the test is completed.

5 Discussion

5.1 Implementability

The test plan generation method proposed in this paper is implementable. First, the *ProcessDependency*, *StartProcess* and *EndProcess* are stored in a database. By means of database query, test plans can be generated easily with the algorithm in Figure 2 in some host languages of the database. It is because the algorithm, in principle, is searching the system architecture, i.e. the *ProcessDependency* and checks the satisfiability of conditions of the searched item with the sets *StartProcess*, *EndProcess* and of course, the test case itself.

5.2 Practicability

We are exemplifying this test plan generation approach in a real life project - the development of a Container Tractor Management System for a major Container Tractor Operation Company in Hong Kong. The company is running a fleet of over 200 container tractors with over 600 staff and contractors. The objective of the system is to manage the container tractor operation, handle the payroll and invoicing. In comparison with traditional approaches that derive test plans directly from specifications manually, we saved much time and effort in deriving the UAT [1]. The time saving is mainly due to the automation of test plan generation as well as the completeness checking of test plans.

5.3 Completeness of Test Plans Generation

An end user oriented test plan is necessary for performing UAT. The classification approach is providing a good foundation for generating the necessary cases that should be tested from the specifications [9, 8, 10, 12, 11, 13, 3, 6, 5, 7, 4]. The test cases generated have no information about the input, output and associated processes, and hence is still have a gap with the applicable UAT plans. The test plan generation method proposed in this paper is bridging this gap. Test plans are generated from the test cases generated from classification approach with reference to the system architecture. The only missing item in the generated test plans are values of some data types such as numeric types. But these values can be generated by some other methods such as partition test or random test [2]. Consequently, the test plans generation method is making the topic of study complete.

[1]Since the project has not been completed at the moment of writing this paper, we do not have a very solid figure on the total amount of time that are saved by using this test plan generation method.

5.4 Limitation

This test plans generation method, inherits the limitation of classification approach test cases generation. Test cases can only be derived from the relations between classes that are described in the specifications. Since test plans are derived from these test cases, the test plans are also limited by this constraint.

Despite the constraint that test plans can only be derived from the relations among classes that have been described in the specification, the test plan generation method is still very useful. It is because the majority of the test plans are testing whether the system developed is conforming with the specifications. The test plans that are beyond the scope of this method are those exception tests. The amount of exception tests, in comparison with specification based normal tests, are small.

6 Conclusion

In view of the need of handling huge amount of test plans in real life UAT, test plans generation can help software developers and end users to shorten the amount of time that are required to develop the UAT plan. Furthermore, test plans generation can also help to ensure the UAT plan is comprehensive. Hence generating test plans automatically is useful, practical and desirable.

Generating test cases by the classification approach is providing a good platform for generating test plans. This approach can generates test cases either based on specifications comprehensively or on demand. More important is that the test cases generated are in the view of end users. The contribution of this paper is providing a method to generate test plans from test cases. The method is implementable and hence can perform the test plan generation automatically.

Our test plans generation method makes the study of the topic test cases generation complete because the method bridges the test cases and the test plans. The automation of test plan generation can help to save huge amount of time in developing test plans from specifications, as well as helping to ensure the completeness of the UAT plans. Consequently the test plan generation method is practical and very useful in real life software testing.

References

[1] C. Berge. *Graphs and Hypergraphs*. North-Holland, Amsterdam, 1973.

[2] Walter J. Gutjahr. Partition testing vs. random testing: The influence of uncertainty. *IEEE Transactions on Software Engieering*, 25(5):661–674, 1999.

[3] Karl R.P.H. Leung and Wai Wong. Generating Test Cases from Class Graphs. In *Proceedings of the 1st Asia-Pacific Conference on Quality Software (APAQS'00)*, pages 285–293, Hong Kong, October 2000. IEEE Computer Society Press.

[4] Karl R.P.H. Leung. Using hypergraph as modeling language for generating test cases on demand. In *Proc. of the 11th Asia Pacific Software Engineering Conference (APSEC 2004)*, pages 510–526, Busan, Korea, December 2004. IEEE Computer Society.

[5] Karl R.P.H. Leung and Wai Wong. Deriving Test Cases Using Class Vectors. In *Proc. of the Seventh Asia-Pacific Software Engineering Conference (APSEC'00)*, pages 146–153, Singapore, 5-8 December 2000. IEEE Computer Society Press.

[6] Karl R.P.H. Leung, Wai Wong, and Joseph K-Y Ng. Generating Test Cases from Class Vectors. *Journal of Systems and Software*, 66(1):35–46, 15 April 2003. ISSN 0164-1212.

[7] Karl R.P.H. Leung, Wai Wong, and Joseph Kee-Yin Ng. A Test Cases Generating Method using Class Vectors. In *Proc. of the IASTED International Conference: Applied Informatics – International Symposium on Software Engineering, Databases, and Applications*, pages 170–175, Innsbruck, Austria, 18-21 February 2002. ACTA Press.

[8] M. Grochtmann and J. Wegener and K. Grimm. Test Case Design Using Classification Trees and the Classification-tree Editor CTE. In *Proceedings of the 8th International Software Quality Week*, pages 1–11, 1995.

[9] M. Grochtmann and K. Grimm. Classification Trees for Partition Testing. *Software Testing, Verification and Reliability*, 3:63–82, 1993.

[10] T.Y. Chen and P.L. Poon. Classification-hierarchy Table: A Methodology for Constructing the Classification Tree. In *Proceedings of the Australian Software Engineering Conference (ASWEC'96)*, pages 93–104. IEEE Computer Society Press, 1996.

[11] T.Y. Chen and P.L. Poon. Improving the Quality of Classification Tree via Restructuring. In *Proceedings of the 3rd Asia-Pacific Software Engineering Conference (APSEC'96)*, pages 83–92. IEEE Computer Society Press, 1996.

[12] T.Y. Chen and P.L. Poon. Construction of Classification Trees via the Classification-hierarchy Table. *Information and Software Technology*, 39:889–896, 1997.

[13] T.Y. Chen and P.L. Poon and T.H. Tse. A New Restructuring Algorithm for the Classification-Tree Method. In S. Tilley and J. Verner, editor, *Proceedings of the 9th International Workshop on Software Technology and Engineering Practice (STEP'99)*, pages 105–114. IEEE Computer Society Press, 1999.

[14] Carolyn Watters and Michael A. Shepherd. A transient hypergraph-based model for data access. *ACM Transactions on Information Systems*, 8(2):77–102, April 1990.