e-Informatica Software Engineering Journal, Volume 4, Issue 1, 2010

Automatic Test Cases Generation from Software Specifications

Aysh Alhroob*, Keshav Dahal*, Alamgir Hossain*
*School of Computing, Informatics and Media, University of Bradford
amhalhro@ bradford.ac.uk, k.p.dahal@ bradford.ac.uk, m.a.hossain1@ bradford.ac.uk

Abstract

A new technique is proposed in this paper to extend the Integrated Classification Tree Methodology (ICTM) developed by Chen et al. [13] This software assists testers to construct test cases from functional specifications. A Unified Modelling Language (UML) class diagram and Object Constraint Language (OCL) are used in this paper to represent the software specifications. Each classification and associated class in the software specification is represented by classes and attributes in the class diagram. Software specification relationships are represented by associated and hierarchical relationships in the class diagram. To ensure that relationships are consistent, an automatic methodology is proposed to capture and control the class relationships in a systematic way. This can help to reduce duplication and illegitimate test cases, which improves the testing efficiency and minimises the time and cost of the testing. The methodology introduced in this paper extracts only the legitimate test cases, by removing the duplicate test cases and those incomputable with the software specifications. Large amounts of time would have been needed to execute all of the test cases; therefore, a methodology was proposed which aimed to select a best testing path. This path guarantees the highest coverage of system units and avoids using all generated test cases. This path reduces the time and cost of the testing.

1. Introduction

Unified Modelling Language (UML) provides diagrams to help the software developer to represent different aspects of design. It has become a standard modelling language for designing software. UML represents the system specifications that could be used in software testing. The common definition of software testing usually refers to the testing of program code and not to the testing of models used in earlier development stages of the software development process, such as requirements engineering, analysis or design. Model testing could identify many faults earlier and could, hence, decrease repair costs. A critical component of testing is the construction of test cases. However, software testing is an expensive and labour-intensive process; typically, testing consumes at least 50% of the total costs involved in developing software [7]. Software testing has two important purposes. First, it is commonly used to expose the presence of faults in software. Second, even if testing does not reveal any fault, it still provides increased justification and confidence in the correctness of the software [18]. The Category Partition Method (CPM) was developed by Ostrand and Balcer [19] to generate test cases from functional specifications using the concept of formal test specifications. Several studies [2, 3] have been conducted which focus on CPM. Recently, Chen et al. [14] enhanced the CPM, by means of their choice relation framework. Based on CPM, Grochtmann and Grimm [16] developed a similar but different method – the Classification Tree Method (CTM). Classification trees have been used to construct test cases in the CTM. The absence of a systematic tree construction algorithm is

the major limitation of this method. As a result, users of this method are left with a loosely defined task of constructing a Classification Tree (*Tu*). For complex specifications, this construction task could be difficult, and hence, prone to human error. If a *Tu.* is incorrectly constructed, the quality of the resultant test cases generated from it will be poorly affected. This problem is solved by Chen et al. [13] who use Integrated Classification Tree Methodology (ICTM). This method helps with the identification of test cases via the construction of classification trees. However, their tree constructed method is rather ad hoc. This results in a variation of classification trees constructed in CTM from one software tester to the next, according to his/her personal experience and expertise. A classification hierarchy table (*Hu*) has been used in ICTM to alleviate this problem. The hierarchy table helps to construct classification trees by capturing the hierarchical relation for each pair of classifications.

In ICTM [13], however, a manual method has been used in order to detect the classifications, associated classes and relationships. The manual process requires more human intermediation which can lead to more errors. Manual extraction of information from the software specifications will also increase the cost of testing. In this paper, we propose an approach for generating test cases automatically from software specifications using a class diagram and representing the software constraints by OCL. After decomposing the specifications to functional units, functional units will be represented as a class diagram. An XML schema mapping technique will then be used to read the specification from the class diagram. Transferring data from the XML to build a hierarchy table *Hu.* will reduce human error in building the table. The construction of the *Hu.* is the step before building the classification tree. The paper also proposes a test data refinement technique to discard duplicate sub-trees. The approaches are based on heuristic techniques for determining appropriate test cases for testing software.

Software testing is used to find as many faults as possible so that a piece of software will work to its maximum capabilities. Path testing is a structural testing method that involves using software units to find every possible executable path. Time and cost are the main factors when testing efficiency, therefore, avoidance of lengthy times in testing was the target set after the legitimate test cases were obtained [1]. This paper extends the work presented in Alhroob, Dahal and Hossain [1] to select the best testing path. This technique offers the best path that covers most of the system units.

The rest of the paper is organised as follows. Section 2 presents the previous work in automatic test data generation for UML. Section 3 presents a methodology to generate test data to test class diagram relationships. Section 4 describes the classification tree concepts, whereas the pruning method of duplicate sub-trees is outlined in Section 5. Section 6 covers the best testing path selection. Finally, conclusions and future works are presented in Section 7.

2. Previous Work

Test data are usually generated from the requirements or the code, while the design is rarely concerned with generating test data. Extensible Markup Language (XML) is used to express the constraints on data and detect the rules of software systems. Bertolino et al. [6] used the XML schema to analyse the system specifications and identify the functional units. Categories are identified from functional specifications. The authors [9] used XML schema mapping and category partition to identify the related constraints and relevant values for each category. In general, the test data generation can be extracted from those values and constraints.

Extracting the information from UML diagrams allows the developer to test the system before writing the code. Heuristic techniques can be applied for creating quality test data. Doungsa-ard et al. [15] proposed a GA-based test data generation technique from specifications. Test cases were generated from sequences of triggers for Unified Modelling Lan-

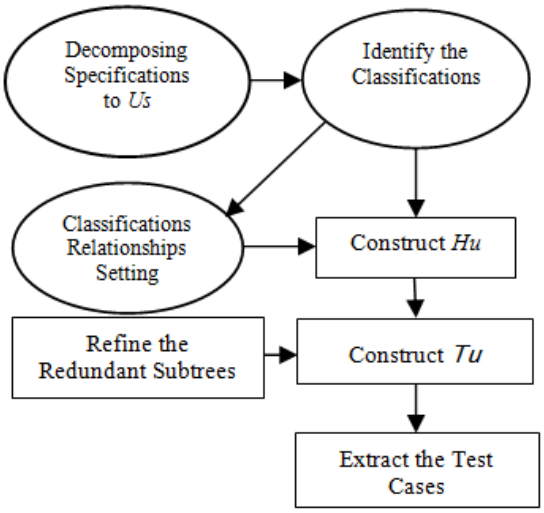

Figure 1. ICTM process

guage (UML) state diagrams. Lia Bao-lin et al. [4] constructed a scenario tree from a sequence diagram. The scenario path was obtained from the tree and the attributes were extracted from the sequence diagram to generate test data automatically. Object Constraint Language (OCL) was used to describe the pre and post conditions for the system to use its system specifications. Sarma et al. [21] proposed a method to extract this information from used case templates, class diagrams and data dictionaries. They also presented an approach to transform the UML sequence diagram to Sequence Diagram Graph (SDG) and provide the SDG with different information necessary to compose test data.

Dehla [23] proposed a technique to generate test data from UML sequence and state diagrams. The main specifications are extracted from the sequence diagram, while the remaining information derives from the state diagram. Sequence diagrams do not provide all information necessary to generate test data automatically. Chen et al. [13] presented ICTM to create test cases from specifications, as shown in Figure 1, via the building of *Hu* and *Tu*. The manual steps, which are indicated by oval shapes in Figure 1, need a software engineering expert. Experts are also needed to decompose the functional unit, identify the classifications and extract the relationships between the classifications. All classifications and their relationships in ICTM are manually entered. The manual processes need more experience, more time and generate higher costs. For large systems there are many classifications and relationships; this requires more effort to input. To avoid the risk and effort, we propose a methodology to enable the manual processes to be done automatically without expert intermediation. Class diagrams will be used to represent the software specifications. The proposed methodology is designed to capture the specifications automatically to build the *Hu* and *Tu*. The building of *Hu* and *Tu* automatically, enables the generation of test cases in an efficient way.

The proposed methodology in this work produced full system coverage test cases, but other issues arose regarding the time needed to execute the test cases, in addition to the cost. Peres et al. [20] introduced a good idea to apply characteristics of software and of testing in test path selection. They used characteristics in path selection strategies, such as complexity, testability and feasibility. They assigned a weight for nodes according to the lower predicate strategy. And the sum of nodes weight was assigned to the branch or path. Emanuela et al. [9] used the degree of similarity between test cases as the main factor for test case selection. This strategy reduced the number of redundant tests and selected the best to execute. Basanieri and co-author [5] proposed a technique to assign

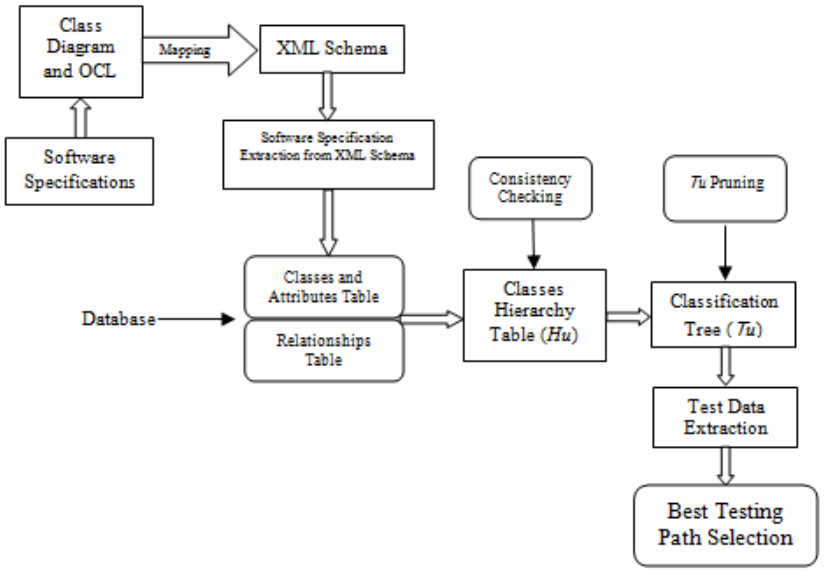

Figure 2. Proposed technique

a weight function to each diagram indicating the functional importance.

Test generation with a verification technology tool [17] extracts the test cases from the UML model. The test cases are selected from a specific objective that a tester would like to test, and can be seen as a specification of a test case. The number of test cases is still large and can be reduced. Our methodology to select the best testing path focused on the maximum coverage percentage and minimum number of test cases. These two factors were treated in previous work [5, 9, 17, 20] separately. In this study, both objectives are considered to achieve the highest results.

3. Methodologies

Class diagram and OCL together represent the functional units. Class diagram mapping to XML code makes the diagram specifications easier to deal with. Due to the variation in design experts, each one will present the functional units using class diagrams in different ways, that is, different class diagrams and different XML codes. To allow the proposed technique to deal with one style of XML, we propose a technique to force different XML to be stored in the same database style.

The specifications will be transferred from the database to construct the *Hu*. The *Hu* will be more reliable due to new consistency checking constraints and automatic information entering. A consistent *Hu* means perfect *Tu*, but control of the number of test cases produced from *Tu* requires more restricted rules to reduce the number of illegitimate test cases. Pruning of the duplication sub-trees will reduce the number of illegitimate test cases. This paper introduces automatic test case generation from software specifications (see Figure 2) and proposes a technique to improve the pruning of sub-trees. There are a large number of legitimate test cases and a technique is needed to select the minimum number to save time and cost. The proposed technique used in this work selects the best path which covers most of the system details.

3.1. UML Class Diagrams

UML class diagrams will be used to represent the component of software specification. The class diagram will represent the functional unit hierarchy relationship. For example, a credit card has two possible types: gold credit card or classic credit card, and each type has its own credit limit, as follows:

– The gold card has two children (credit limit of \$5000] and credit limit of \$6000].

– The classic card also has two children (credit limit \$2000] and credit limit \$3000].

Figure 3 represents the above functional units using three hierarchy classes. OCL is used to represent the constraints and determine the relationship between the attributes in the main class with sub-classes.

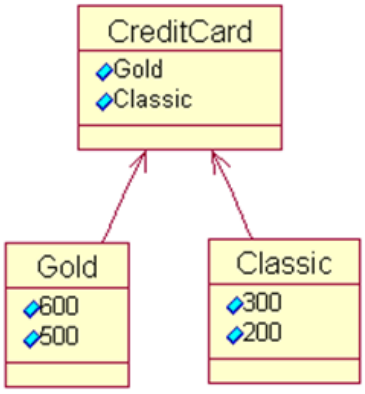

Figure 3. Example of functional unit representation by class diagram

XML mapping is used to extract the software specifications and capture the elements of software from the UML diagram. UML is a standard design modelling language and XML is widely being accepted as an information representation and sharing language across the Internet; efforts have been initiated to map UML diagrams to XML documents [22].

3.2. Automatic Detection for System Specifications

The first three phases of ICTM in Figure 1 are manual phases and the construction of *Hu* is dependent on the relationship setting. A class diagram as a software system model is used to decompose the specification and identify the relationships automatically. Classes, attributes and relationships for the class diagram can be extracted through mapping the diagram to XML. In the proposed approach the XML structure depends on the class diagram design. The diagram design depends on the designer's view, so there is a probability of extracting different XML schema for the same system specifications. Standard XML schema is not our concern, but the specifications in that schema must be captured in a standard way. The extracted classes and attributes will be stored in proper database style, like Table 1. The relationships between classes can be stored in a different table. Now, an automatic transfer technique will be used to transfer

Table 1. Classes and attributes for class diagram in Figure 4

Class	Att1	Att2	Att3
A	$a1$	$a2$	
B	$b1$	$b2$	
C	$c1$	$c2$	
D	$d1$	$d2$	
E	$e1$	$e2$	
F	$f1$	$f2$	
G	$g1$	$g2$	
H	$h1$	$h2$	$h3$
I	$i1$	$i2$	

data from database tables to *Hu*, instead of manually inserting. To make the proposed approach clearer, the following case will be used to represent the main steps of the methodology. Suppose a software tester is given the following of a program *arith-sum*:

1. *arith-sum* has nine input variables A, B, C, D, E, F, H, and I.
2. H has three possible values (denoted by $h1$, $h2$, and $h3$), whereas each of the remaining variables has two possible values (denoted, for example, by $a1$ and $a2$ for A).
3. The input domain of *arith-sum* may contain any combination of possible values from some of these variables, except the following:
   – (A is $a2$) and (B is $b1$ or $b2$)
   – (A is $a2$) and (C is $c1$ or $c2$)
   – (A is $a2$) and (D is $d1$ or $d2$)
   – (A is $a1$) and (E is $e1$ or $e2$)
   – (B is $b2$) and (C is $c1$ or $c2$)
   – (B is $b2$) and (D is $d1$ or $d2$)
   – (B is $b1$ or $b2$) and (E is $e1$ or $e2$)
   – (C is $c2$) and (D is $d1$ or $d2$)
   – (C is $c1$ or $c2$) and (E is $e1$ or $e2$)
   – (C is $c1$ or $c2$) and (F is $f2$)
   – (C is $c1$ or $c2$) and (H is $h1$, $h2$, or $h3$)
   – (D is $d1$ or $d2$) and (E is $e1$ or $e2$)
   – (D is $d1$ or $d2$) and (F is $f2$)

- (D is $d1$ or $d2$) and (H is $h1$, $h2$, or $h3$)
- (E is $e1$ or $e2$) and (G is $g1$ or $g2$)
- (F is $f2$) and (G is $g1$ or $g2$)
- (F is $f1$) and (H is $h1$, $h2$, or $h3$)
- (G is $g1$ or $g2$) and (H is $h1, h2$, or $h3$)

4. *arith-sum* calculates the arithmetic sum of those variables entered.

Suppose we simply define the classes as the input variables and the attributes as the possible values. For example, A is taken as a class with $a1$ and $a2$ as its attributes. Then Figure 4 shows the class diagram for *arith-sum*.

3.3. Classes Hierarchy Table

Automatic construction of a classification hierarchy table, *Hu* with class relationships for each pair of classes is the main target in this section. There are four possible types of hierarchical relationships, as follows [10, 13]:

1. Class $[X]$ is a loose ancestor of class $[Y]$ (denoted by $[X] \Leftrightarrow [Y]$.
2. Class $[X]$ is a strict ancestor of $[Y]$ (denoted by $[X] \Rightarrow [Y]$). A black arrow means direct relation, but red indicates an indirect relation.
3. Class $[X]$ is incompatible with Class $[Y]$ (denoted by $[X] \sim [Y]$).
4. Class $[X]$ has other relations with Class $[Y]$ (denoted by $[X] \bigotimes [Y]$).

The conditions associated with each of the above hierarchical relations are commonly exclusive and exhaustive. These hierarchical relations are used to determine the relative position of $[X]$ and $[Y]$ in *Tu*. For example, $[X] \Rightarrow [Y]$ corresponds to the situation where $[X]$ will appear as either a parent or an ancestor of $[Y]$ in *Tu*; in current work the loose ancestor relationship was discarded. Figure 5 depicts the completed *Hu* Every element in it contains a hierarchical operator and corresponds to the hierarchical relations between a pair of classifications.

3.4. Consistency Checking

Cain et al. [8] introduced the consistency problem and proposed a technique to detect inconsistency relations. Let t_{ij} denote the element at the i^{th} row and the j^{th} column of Figure 5. Consider t_{12} and t_{21} in Figure 5. They correspond the $[A] \Rightarrow [B]$ and $[B] \bigotimes [A]$, respectively. Suppose,

- t_{21} constraints are entered before that for t_{12},
- t_{21} constraints are entered correctly, causing "$\bigotimes$" to t_{21} to assign the hierarchical operator,
- a mistake has been made during the entry of the constraints for t_{12}, causing incorrect assignment of the hierarchical operator "$\sim$" to t_{12}.

As we note that the error is unwanted, to recover this problem, a methodology is proposed in this paper to ensure that all of the relationships are entered in a systematic way and no conflict occurs between them. We need the following five conditions to do that:

1. For e_{ij} (where e_{ij} is corresponding for all classes) we have to detect the relationships automatically for each pair of classes (A, B) in *Hu*.
2. $X_i \rightarrow Y_i$, where X and Y are two associated classes in the database.
3. If $A = X$ and $B = Y$, then A is a child of B, and all of the attributes of A are related with at least one of the B attributes.
4. $A = X$ and $B \neq Y$, then A is incompatible with Y.
5. If $A = Y$ and $B = X$, then A is a parent of B, and at least one of the A attributes are related with all B's attributes.

4. Classification Tree (*Tu*)

Based on a predefined tree construction algorithm [13], the corresponding *Tu* can be automatically constructed from the *Hu* in Figure 5. The tree represents the relationships between the classes and determines the parents and children of classes. The classification tree is used to generate the test cases; if the test cases cover 100% of the tree branches that means all parents, children and attributes will be tested. Complete or legitimate test case extraction is our target. Human error in the specification ex-

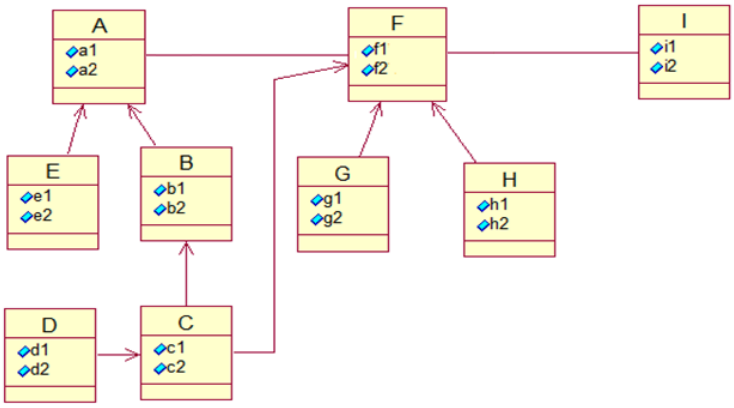

Figure 4. Class diagram for *arith-sum*

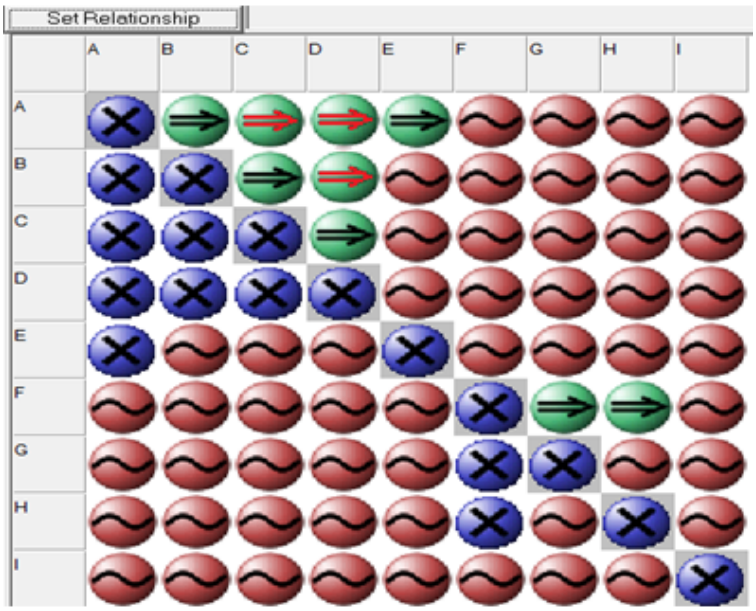

Figure 5. Classification-Hierarchy table (*Hu*)

traction phase is avoided by using the proposed automatic methodology.

Occasionally, a classification tree may not be able to reflect all the constraints between classifications. Therefore, all potential test cases constructed from the classification tree should be verified with the specification. In this way, we can classify and remove the potential test cases that deny the specification. Such potential test cases are known as illegitimate test cases. Chen and Poon [11] proposed that the final purpose of the classification tree method is to construct legitimate test cases, and the classification tree is just a means for this construction. Given a classification tree *Tu*, let N^i and N^t be the number of potential test cases and legitimate test cases, respectively. Chen et al. [12] defined an effectiveness metric, E_p for *Tu* as the follows:

$$E_P = \frac{N^t}{N^i} \tag{1}$$

For more illustration, for equation (1), let $N^i = 40$ and $N^t = 5$, then $E_p = 0.125$. N^t can only be known after removing all illegitimate test cases from the set of possible test cases. Obviously, a small value of E_p is undesirable, as effort will be wasted on illegitimate test cases. The existence of duplicate sub-trees under different top-level classifications in a classification tree is a main cause of a poor E_p.

From this remark, Chen and Poon developed a tree restructuring algorithm to remove duplicates, to obtain a better value of E_p for classification trees with duplicate sub-trees under different top-level classifications [12]. Deleting the duplicate sub-tree will cause the illegitimate test cases to be discarded and will increase the effectiveness metric of the classification tree. Determining the duplication availability and choosing a suitable sub-tree to delete are main factors for duplication sub-tree pruning.

5. Pruning the Duplication Subtrees

Chen et al. [12] proposed an algorithm to avoid duplicate sub-trees and improve the value of E_P. They observed that the algorithm for removing duplicates has many limitations, such as dealing with duplicate sub-trees under different top-level classifications and the fact that only one set of duplicate sub-trees can be removed from the classification tree at any one time. To overcome these limitations, we propose a restructuring algorithm for pruning the duplicate sub-tree, hence, improving the value of E_p. The proposed algorithm will deal with duplicate sub-trees and choose the best ones to keep with Tu and remove the others. The algorithm will compare every sub-tree (ST) in Tu with others to detect duplications in same and different top-level classifications (P_i). The duplicate sub-tree suitable for deleting is one that produces a large number of test cases by integrating with others under the same or different top level. The following methodology illustrates the detection and deletion of suitable duplicate sub-trees.

1. P_1, P_2 and P_3 are top level class, for example, they correspond to A, F and I respectively as shown in Figure 6, where $P_i \geq 1$.
2. The Tu has been divided into levels L_1, L_2,..., L_i, where $L_i \geq 1$. Every L_i has a value depending on the level number, for example L_1 has a value 1, L_2 has a value 2, etc.
3. Every classification (X) in Tu has its own principle value sequentially, for example ($A = 1$, $B = 2$,..., $X_n = V$). Each children (x) for X will take the same value of X. $x_v = L_i * X_n$, where x_v is the value of each child.
4. We propose (2), (3) and (4) equations to calculate some of parameters values to determine which trees in the system must be deleted.

$$X_V = \sum_{n=1}^{n} x_v \tag{2}$$

$$Q_P = \sum_{n=1}^{n} X_v \tag{3}$$

$$Q_{ST} = \sum_{n=1}^{n} X_{Dv} \tag{4}$$

$$RQ_{ST}(\textit{Ratio of duplication } ST \textit{ in } P_i) = \frac{Q_{ST}}{Q_P} \tag{5}$$

Where X_v, Q_P, Q_{ST} and X_{DV} are the values of X, P_i, duplicate ST and duplicate classes, respectively. We observe that if the ratio of Q_{ST} on Q_P in equation (5) is smaller, then the number of branches in P_i be big, which means more test cases will be generated from this P_i, If we delete the duplication ST from P_i which has more branches, we avoid generating more illegitimate test cases. For example, if RQ_{ST} in P_1 < RQ_{ST} in P_2 the duplication ST in P_1 must be deleted. The algorithm repeats the above process until there are no duplicated STs across any pairs of distinct top-level sub-trees. This algorithm deals with all duplicated STs in Tu whether in the same top level or different and treats duplications for two sub-trees or more. By referring to Figure 4, the classification tree in Figure 6 contains duplications for C ST. The C sub-tree arises in two places, the first one occurs under the B class and the second one under the F class; the algorithm will detect the duplica-

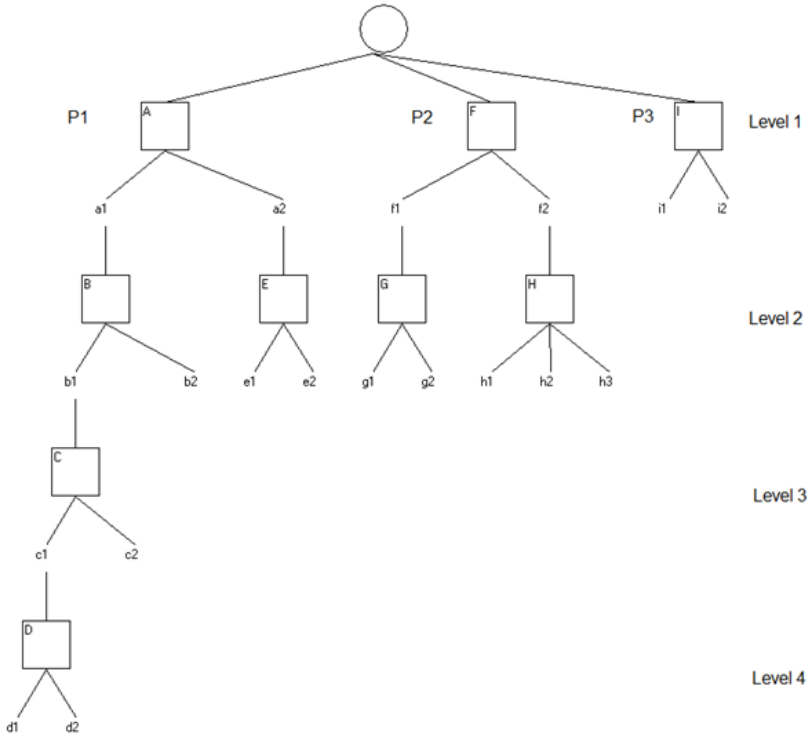

Figure 6. *arith-sum* Classification Tree

tion by comparing classifications with others in Tu even in different top level classes. If the duplication is detected, the algorithm starts to capture the suitable ST to delete. For example, as noted in the class diagram the C class is associated with B and F; that means that the C sub-tree should appear under two parents of classes. Duplication will occur even under B or F. One of the sub-trees must be chosen for deletion depending on the proposed sub-tree pruning algorithm.

In this case, and by referring to equation (4), the duplicate sub-tree that comes under F is selected for deletion. From Tu of *arith-sum* in Figure 6, a total of 60 potential test cases can be constructed; some of the test cases are shown in Figure 7. The total number of test cases, before deleting the duplicate sub-trees, was 108, so the classification tree pruning technique deleted 48 illegitimate test cases by checking the 60 legitimate potential test cases against the specification of *arith-sum*. 32 potential test cases were found to be illegitimate and therefore removed. For example, the potential test cases 5–10 were illegitimate because class $(F) = f2$ cannot coexist with class $(C) = c1$ and $c2$.

6. Best Testing Path Selection

Testing is the process of executing a system with the intention of finding errors. Assume that there are 5 possible paths with loop<10, its equal 10^7 different execution flows. If we executed one test per millisecond, it would take 1.585 years to test this system. The proposed methodology in this work aims to select a best testing path. This path guarantees the highest coverage of system units. Each test case generated for the class diagram in Figure 3 represents a test path. Before continuing to explain the technique, we have to differentiate between the test path and the case. The test case is the combination of the node attributes, in contrast, the test path is the nodes that are shared in a test case, i.e. Test case 2 in Figure 7 is $A = a1$, $B = b1$, $C = c1$, $D = d1$, $F = f1$, $G = g1$, $I = i2$ and the test path 2 is A, B, C, D, F, G, I.

Set Relationship | Classification Tree | Potential Test Frames

Frame 1	A = a1	B = b1	C = c1	D = d1	F = f1	G = g1	I = i1
Frame 2	A = a1	B = b1	C = c1	D = d1	F = f1	G = g1	I = i2
Frame 3	A = a1	B = b1	C = c1	D = d1	F = f1	G = g2	I = i1
Frame 4	A = a1	B = b1	C = c1	D = d1	F = f1	G = g2	I = i2
Frame 5	A = a1	B = b1	C = c1	D = d1	F = f1	H = h1	I = i1
Frame 6	A = a1	B = b1	C = c1	D = d1	F = f1	H = h1	I = i2
Frame 7	A = a1	B = b1	C = c1	D = d1	F = f1	H = h2	I = i1
Frame 8	A = a1	B = b1	C = c1	D = d1	F = f1	H = h2	I = i2
Frame 9	A = a1	B = b1	C = c1	D = d1	F = f1	H = h3	I = i1
Frame 10	A = a1	B = b1	C = c1	D = d1	F = f1	H = h3	I = i2

Figure 7. Some of potential test cases generated from *Tu arith-sum* (10 out of 60)

The best testing path technique, which covers maximum units, concerns two main aspects.

Firstly, the weight of each class (node) is dependent on the number of attributes and its level. The level of node will be affected by the number of participant nodes and branches, i.e. H node in Figure 6 has a weight of 12. The weight of H has been calculated by equation (2) and $x_v = L_i * X_n$, where x_v is the value of each child. Secondly, the weight of each path is the weight of all nodes that share one path, i.e. test case 1 in Figure 7 goes through A, B, C, D, F, G and I nodes. The weight of each node is as follows: $A = 1, B = 4, C = 6, D = 8, F = 4, G = 8$ and $I = 6$. The weight of the path is calculated as follows:

$$Pw = \sum_{n=1}^{n} X_v \tag{6}$$

where is the Pw is the weight of path and X_v is the weight of each node. One of the case studies used in this work is the ATM machine. This system contains 18 nodes and each node has its own attributes. Table 2 represents each node weight that has been calculated automatically. As noted, the nodes are called by their numbers and not by names, because the testing path selection methodology deals with nodes by their numbers, i.e. CardReader = 1, Inquiry = 2, Deposit = 3, ... etc.

Figure 8 represents the ATM system nodes; the Node Tree (NT) shows the hierarchy relationships between the nodes. The tree represents the relationships between the classes and determines the test paths. If the test paths cover 100% of the tree nodes that means all system units will be tested.

Table 2. ATM Nodes Weight

N_i	Weight	N_i	Weight
1	2	10	60
2	12	11	66
3	18	12	96
4	32	13	130
5	20	14	112
6	48	15	225
7	14	16	160
8	32	17	68
9	54	18	36

In this phase, the proposed methodology extracted the test paths automatically based on the proposed test path construction algorithm. 15 test paths were generated from NT (see Table 3), one is the best one. It is not necessary that the testing path covers all units of the system; any test path that covers the highest percentage of systems is the best one. The highest percentage does not mean the largest number of nodes. Each node has its own weight, as illustrated in Table 2. The best path, that has the highest weight out of the total weight, is P_9, as shown in Table 3. P_9 covers 10 nodes out of 18 with 51% system details coverage.

In fact, one test path cannot cover all units, as there may be many paths/loops. To improve the coverage of the system, we propose selecting the second best testing path as well to support the first best testing path. The selection method for the second best testing path depends on the non-similarity of nodes contained in the

Table 3. Testing Paths Weight(TPW)

P_i	N_1	N_2	N_3	N_4	N_5	N_6	N_7	N_8	N_9	N_{10}	TPW
P_1	1	2	3	4	7	8	9	18	–	–	200
P_2	1	2	3	5	7	8	9	18	–	–	188
P_3	1	2	3	6	7	8	9	18	–	–	216
P_4	1	2	3	4	7	10	11	12	13	18	466
P_5	1	2	3	5	7	10	11	12	13	18	454
P_6	1	2	3	6	7	10	11	12	13	18	482
P_7	1	2	3	4	7	10	11	14	15	18	577
P_8	1	2	3	5	7	10	11	14	15	18	565
P_9	1	2	3	6	7	10	11	14	15	18	593
P_{10}	1	2	3	4	7	10	11	14	16	18	512
P_{11}	1	2	3	5	7	10	11	14	16	18	500
P_{12}	1	2	3	6	7	10	11	14	16	18	528
P_{13}	1	2	3	4	7	10	17	18	–	–	242
P_{14}	1	2	3	5	7	10	17	18	–	–	230
P_{15}	1	2	3	6	7	10	17	18	–	–	258

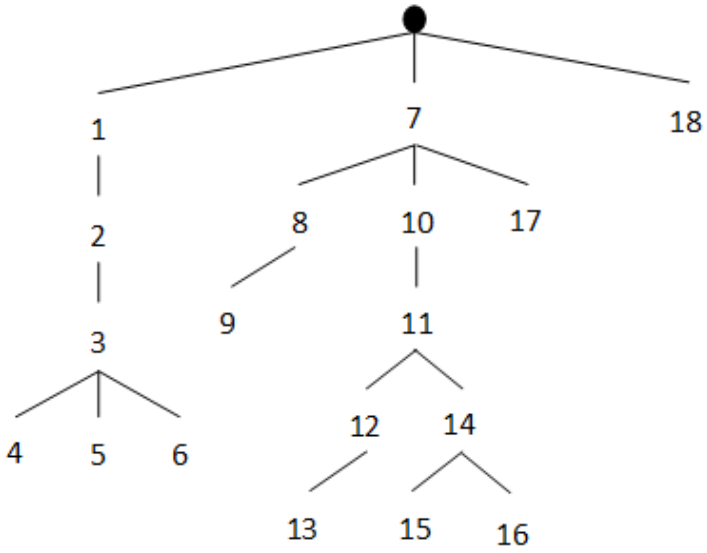

Figure 8. ATM Nodes Tree

best testing path. In other words, we want to select a second best testing path, which contains as many different nodes as possible compared to the first best testing path. Emanuela et al. [12] used the similarity function to reduce the test cases. To select the second best path, we used node non-similarity criterion. Based on the non-similarity degree between the best path and others, the testing paths eliminated were those with the biggest similarity degree. There is a probability for getting more than one test path with the same degree of non-similarity criterion; the highest weight of non-similar nodes is the factor used to choose one of them.

In ATM testing path there are four paths (P_1, P_2, P_4 and P_5) met the highest non-similar criterion, non-similar path with the highest

Table 4. Non-similar paths weight

P_i	Non-similar Nodes	Weight
P_1	4, 8 and 9	118
P_2	5, 8 and 9	106
P_3	6, 8 and 9	134
P_4	4, 12 and 13	285

nodes weight is selected. Table 4 represents those paths with non-similar node weights. P_3 has two conditions necessary (non-similarity and highest weight) for being chosen as the second best testing path. Both testing paths (best and second best testing path) cover more than 74% of system details.

7. Conclusion

In this paper the ICTM has been improved to detect specifications automatically. These specifications are used to generate test cases. To control the consistency of relationships in *Hu*, we proposed an algorithm to enter the class hierarchy relationships in a systematic way. Consistency relationship entering techniques support the reliability of ICTM.

In this paper, a restructured algorithm was proposed to remove duplication sub-trees, either at the same top level or at different top levels. This technique can offer more pruning and produce legitimate and non-duplicated test cases. Testing path selection was one of the concerns in this work in order to reduce the number of expectation flows. Test paths have been determined and one has been selected automatically to be the best among them. The best testing path covers most of the system units and avoids the undesirable time needed to execute all test paths. To improve the percentages of coverage, we propose the selection of additional test paths based on dissimilarity to the best test path.

In future work we will conceder to use class diagrams, OCL and sequence diagrams to represent software specifications to provide other additional information. Therefore, by combining these two UML specifications in future work we will be able to capture most of the system specifications.

References

[1] A. Alhroob, K. Dahal, and A. Hossain. Automatic test cases generation from software specifications modules. In *Proceedings of the 4th IFIP TC2 Central and East European Conference on Software Engineering Techniques*, pages 130–142. Springer, 2009.

[2] N. Amla and P. Ammann. Using Z specifications in category partition testing. In *Systems Integrity, Software Safety and Process Security: Building the System Right*, pages 3–10, Gaithersburg, MD, USA, IEEE Press, 1992.

[3] P. Ammann and J. Offutt. Using formal methods to derive test frames in category-partition testing. In *Computer Assurance, 1994. COMPASS '94 Safety, Reliability, Fault Tolerance, Concurrency and Real Time, Security. Proceedings of the Ninth Annual Conference on*, pages 69–79, Gaithersburg, MD, USA, IEEE, 1994.

[4] L. Bao-Lin, L. Zhi-shu, L. Qing, and C. Y. Hong. Test case automate generation from UML sequence diagram and OCL expression. In *Proceedings of the 2007 International Conference on Computational Intelligence and Security: CIS*, pages 1048–1052, 2007.

[5] F. Basanieri, A. Bertolino, and E. Marchetti. The Cow_Suite approach to planning and deriving test suites in UML projects. In J.-M. Jézéquel, H. Hussmann, and S. Cook, editors, *UML 2002–the Unified Modeling Language*, pages 383–397. Springer, 2002.

[6] A. Bertolino, J. Gao, E. Marchetti, and A. Polini. Automatic test data generation for XML schema-based partition testing. In *Proceedings of the Second International Workshop on Automation of Software Test*, page 4. IEEE Computer Society, 2007.

[7] B. Boris. *Software testing techniques*. Van Nostrand Reinhold Co, second edition, 1990.

[8] A. Cain, T. Y. Chen, D. Grant, P. L. Poon, S. F. Tang, and T. H. Tse. An automatic test data generation system based on the integrated classification-tree methodology. *Software Engineering Research and Applications*, pages 225–238, 2004.

[9] E. G. Cartaxo, F. G. O. Neto, and P. D. L. Machado. Automated test case selection based on a similarity function. In *Workshop Modellbasiertes Testen (MOTES07)*, Bremen, 2007.

[10] T. Y. Chen and P. L. Poon. Classification-hierarchy table: a methodology for constructing the classification tree. In *Proceedings of the 1996 Australian Software Engineering Conference*, page 93, Washington, DC, USA, IEEE Computer Society, 1996.

[11] T. Y. Chen and P. L. Poon. Improving the quality of classification trees via restructuring. In *Proceedings of the Third Asia-Pacific Software Engineering Conference*, page 83, 1996.

[12] T. Y. Chen, P. L. Poon, and T. H. Tse. A new restructuring algorithm for the classification-tree method. In *Proceedings of the Software Technology and Engineering Practice*, pages 105–114, 1999.

[13] T. Y. Chen, P. L. Poon, and T. H. Tse. An integrated classification-tree methodology for test case generation. *International Journal of Software Engineering and Knowledge Engineering*, 10(6):647–679, 2000.

[14] T. Y. Chen, P. L. Poon, and T. H. Tse. A choice relation framework for supporting category-partition test case generation. *IEEE transactions on software engineering*, 29(7):577–593, 2003.

[15] C. Doungsa-ard, K. Dahal, A. Hossain, and T. Suwannasart. *Advanced Design and Manufacture to Gain a Competitive Edge*, chapter GA-based for Automatic Test Data Generation for UML State Diagrams with Parallel Paths, pages 147–156. Springer, London, 2008.

[16] M. Grochtmann and K. Grimm. Classification trees for partition testing. *Software Testing, Verification and Reliability*, 3(2):63–82, 1993.

[17] C. Jard and T. Jéron. TGV: theory, principles and algorithms. *International Journal on Software Tools for Technology Transfer (STTT)*, 7(4):297–315, 2005.

[18] K. W. Miller, L. J. Morell, R. E. Noonan, S. K. Park, D. M. Nicol, B. W. Murrill, and J. M. Voas. Estimating the probability of failure when testing reveals no failures. *IEEE transactions on Software Engineering*, 18(1):33–43, 1992.

[19] T. J. Ostrand and M. J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[20] L. M. Peres, S. R. Vergilio, M. Jino, and J. C. Maldonado. Path selection in the structural testing: Proposition, implementation and application of strategies. In *Proceedings. XXI Internatinal Conference of the Chilean Computer Science Society*, pages 240–246. SCCC, 2001.

[21] M. Sarma, D. Kundu, and R. Mall. Automatic test case generation from UML sequence diagram. In *Proceedings of the 15th International Conference on Advanced Computing and Communications*, pages 60–67, Washington, DC, USA, 2007. IEEE Computer Society.

[22] J. Singh. Mapping UML diagrams to XML. Master's thesis, Jawaharlal Nehru University New Delhi, India, 2003.

[23] D. Sokenou. Generating test sequences from UML sequence diagrams and state diagrams. *Informatik für Menschen*, 2(94):236–240, 2006.