5th World Congress for Software Quality – Shanghai, China – November 2011

Test Case Extraction and Test Data Generation from Design Models

Xiaojing Zhang
NTT Cyber Space Laboratories
Yokosuka, Japan
zhang.xiaojing@lab.ntt.co.jp

Takashi Hoshino
NTT Cyber Space Laboratories
Yokosuka, Japan
hoshino.takashi@lab.ntt.co.jp

Abstract

Comprehensive testing is essential to ensure that software works correctly. Improving this process is necessary for software quality assurance. Beside “test execution”, the tasks of “test design” include creating the test cases and the test data. Test design is important but is time and effort consuming when done manually. This paper proposes a method that generates test design artifacts from UML design models; it is intended to improve SUT coverage and test density with minimal manual labor input. We implement a prototype and perform a case study to compare automatic generation and manual creation.

1. Introduction

1.1 Background

The economic loss caused by software defects has become a significant object of public concern. In this study, we aim to achieve better quality assurance by improving the testing process by striking the optimum range between quality and cost. We consider testing as the confirmation of whether the product is developed as intended, rather than just a search for bugs. Since the intentions of the customer are formalized into software design, the subject of our study is testing to confirm “whether the product is implemented as it was designed”.

To achieve adequate test coverage, it is essential to increase the quantity and quality of the test, in addition to related factors, test management for example. We use “test density” as a metric of test quantity. It can be represented by the following formula where SUT represents software under test:

Test density = Number of test cases / Size of the SUT.

“SUT coverage” can be used as a metric of test quality. Here, we consider two kinds of SUT coverage, structure coverage and input space coverage. Structure coverage is the percentage of SUT elements that have been tested. This can be represented as:

Structure coverage = Number of elements been tested / Total number of elements.

Here, the elements that make up the structure of the software include instructions, branches, and paths, to methods, classes, and components. In this study, we focus on path coverage. Input space coverage, is the percentage of SUT inputs that have been tested. This can be represented as:

Input Space Coverage = Inputs used for testing / Entire input space.

Here, the entire input space is the set of all the possible inputs that can be given to the SUT. Notice that both denominators of structure coverage and input space coverage could be infinite, but they should somehow be reduced to finite numbers to get an adequate approximation of coverage. For example, using equivalence classes to choose inputs is a good idea for quality assurance, because the number of inputs is usually infinite but we can only perform testing a finite number of times.

Although it is ideal to get high test density and SUT coverage, the cost of the additional manpower needed would become a problem in almost every project. To achieve the goal within realistic constraints on cost and delivery time of development, testing with low labor input is required strongly.

For test laborsaving, our study focuses on test design automation. Test design consists of two activities: extracting test cases and creating test data from the software design. The test data materializes a test case and makes it executable. Unlike test execution, test design can only be done by a few skilled engineers, so it may become the “bottleneck” of the test process. In this study, our goal is to increase the test density and SUT coverage while reducing the man-hours required for testing by means of test design automation.

1.2 Motivation and Contributions

We propose a method to automatically generate test cases and test data as test design artifacts, based on the software design described in UML (UML model). In particular, the inputs of this generation are UML 2.0 class diagrams and activity diagrams, and outputs are test cases, and test data with hierarchical structure such as object data-type. We had three reasons for choosing this approach.

First, familiar notations. Existing research on test design automation especially test case extraction, as mentioned in the literature [1, 2, 3], was mainly aimed at embedded systems and communications protocols which are based on state transition models, so the scope considered is still limited. Moreover, remember that the input of test design is software design, which is the formalized intention of users and developers. One problem is that software design notation is often “original” and “esoteric”, which creates a serious barrier to the technique's practitioners. To achieve usability, our study adopts UML which has become widespread in recent years as a notation for software design; the resulting activity diagrams and class diagrams are well known to developers. UML 2.0 is well suited for the formalization of software design information and it also has a high affinity with automated processing.

Second, software design as input. As a part of test design, considering the variation in test data needed and then creating the data instances themselves is very time-consuming if you want to increase SUT coverage and test density. This is because a huge amount of test data is needed, and before creating the data you have to consider the selection of input values, which should cover the input space comprehensively and efficiently. In contrast, existing research on test data generation, as surveyed in the literature [4, 5, 6], has the following problems. A lot of studies generate test data from source code, but few studies have considered using software design as input. Some tools [7] can generate pseudo personal information such as combinations of name and address, etc. These kinds of data are useful but they are randomly generated from a dictionary, and do not reflect design information. Unlike software design, source code and dictionaries might be different from the user/developer's intention, so the validity of the resulting content has to be verified.

Third, structured test data can be handled. With regard to the data types that can be generated, the great majority of studies dealing with only primitive types for example integer and Boolean. No fully developed approach is known to cover all data types. We consider that test data generation must yield test data that has complicated structure. Therefore, this study focuses on test data generation method that can output hierarchical structured data (e.g., object data-type). Data-types such as XML and objects are often used in recent software, so using only primitive data type such as numerical values and the character strings as the test data is insufficient.

This paper makes the following contributions: First, we present a method that achieves higher test density and SUT coverage by automatically generating test cases and test data from UML design models. It can handle data-types with hierarchical structure, and it can generate abnormal test data so it is more useful in real projects. Second, we evaluate our method on a Java web system, and extract some useful conclusions including the correspondence and differences between manual creation and automatic generation.

The remainder of this paper is organized as follows. Section 2 describes test case and test data generation method. Section 3 evaluates our method by a case study. Section 4 shows future works and concludes the study.

2. Generation Method

2.1 Test case extraction

IEEE Standard [8] defines test case as below:

A set of test inputs, execution conditions, and expected results developed for a particular objective, such as to exercise a particular program path or to verify compliance with a specific requirement.

In this study, we consider a test case to be a pair of a particular “execution path” within the intended behavior, and the “test data” including the inputs and the expected results. Figure 1 outlines the proposed generation method.

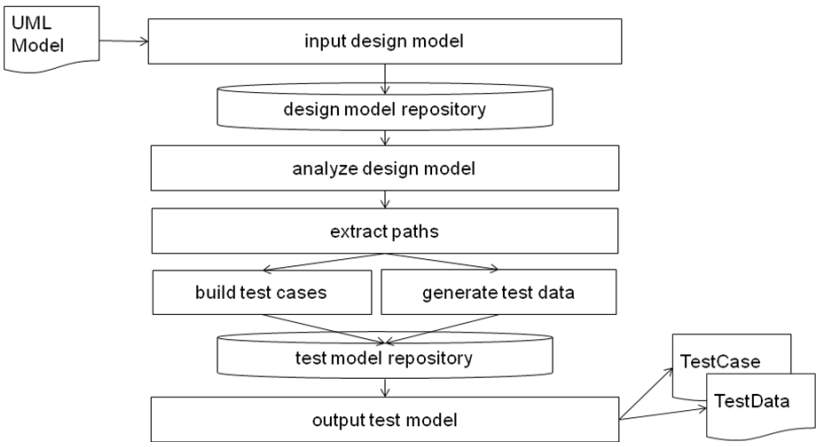

Figure 1 Outline of proposed generation method

2.1.1 Execution path

First, we analyze the structure of the input UML design model and decompose it into components, classes, methods and activities. We then extract the execution paths, each of which is a directed graph, from each activity by a simple depth-first search algorithm. This is a search for the final node starting from initial node.

In extracting the execution paths, all possible execution paths are extracted exhaustively, to ensure adequate structure coverage.

However, when a loop exists in the activity, infinite execution paths can be extracted. We convert each loop into two paths: loop by-pass and single loop execution. The number of paths extracted can be represented as the sum of permutations shown below, when *N* is the number of loops within the activity: Number of paths = summation of (*N* P *i*), for *i* = 0 to *N*.

2.1.2 Test data

The other element of the pair, the test data, is necessary to create executable test cases. Therefore, for each extracted execution path, we have to generate the test data that can trace this path; the procedure for this is described in the next section.

2.2 Test data generation

The test data generation method ensures adequate SUT coverage, especially input space coverage. In order to generate the test data with high coverage, we need to obtain all factors and levels that constitute the test data. In software testing, the factors refer to the variables to be considered, while the levels mean the values set to a specific variable. For instance, there are two levels (male and female) for the factor of sex.

The test data is generated with the 4 steps explained below, and an UML model (class diagram, activity diagram only) is assumed to be the input of this.

2.2.1 Extraction of the factors

We acquire the activity parameter nodes which are the input parameters of the activity, and these are assumed to be factors. Each input parameter has a name and type.

2.2.2 Generation of the test data specification

The test data specifications are a set of constraints that the test data should follow in order to trace one execution path of SUT. A test data specification is generated for each factor of each execution path. In other words, it contains the information of “what variable x should be to suit path y?”

(a) Data structure

The test data specification maintains the domain of the factor internally. Figure 2 shows the structure of the test data specification.

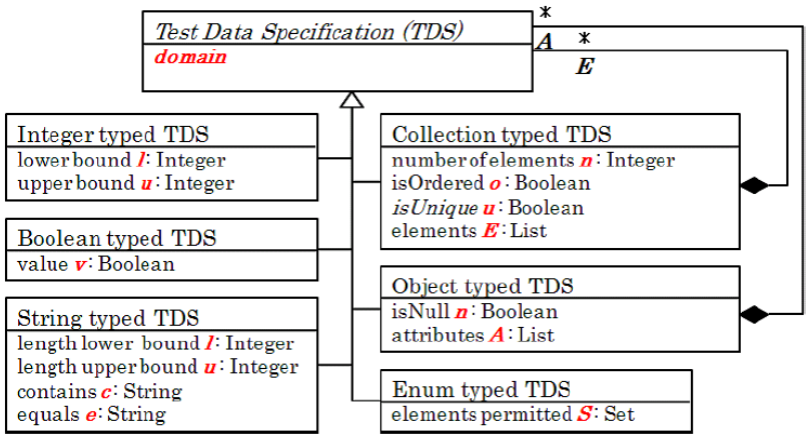

Figure 2 Test data specification

We enable the generation of the test data with hierarchical structure, by defining the test data specification to match the factor of object data-type. As shown in Figure 2, the "object typed test data specification" contains list *A* in its domain, and the list consists of the test data specifications of its own attributes. As a result, we can recursively specify the constraints that the object data-type factors should satisfy, which allows us to handle hierarchical structures. Moreover, for collection type, we consider that it is possible to handle the repetition structure by preserving the test data specification for each element in the collection.

The domain's form of each test data specification is different and depends on the data-type of the factor. For instance, the factor of the integer data-type is associated with the Integer typed test data specification, and this specification has a domain defined by two parameters of lower bound *l* and upper bound *u*. In addition, each kind of domain has an invariant condition. An invariant condition is a constraint that the domain should always satisfy. For instance, "*l* <= *u*" should always be true in the integer typed test data specification.

(b) Extraction

Extracting the test data specification proceeds in 2 steps: initialization and update.

Initialization is explained as follows. For one specified factor on a certain execution path, we acquire the "type" of the factor according to the UML Meta model. Next, a test data specification that matches the type of the factor is generated. For example, the string typed test data specification is generated for a String data-type factor. Next, an initial value is set to the domain of the generated test data specification. For instance, lower bound *l* and upper bound *u* in the Integer typed test data specification will be set to MIN and MAX value of the system default. However, if the data definition concerning the type of the factor is described as an invariant condition in the UML model (constraints associated to the class in the class diagram), we give higher priority to this data definition and use it as the initial value of the domain.

In the update step, we clarify the conditions in order to trace down the execution path, and update the test data specification. The execution path is traced from the initial node to the final node, and the preconditions of the activity, the local preconditions and the guard conditions in the execution path, are processed sequentially. For instance, assume that when scanning through a certain execution path, we find a guard condition that states "name.length < 20". This condition refers to the factor "name", so we update (overwrite) length upper bound *u* to 19, in the domain of the test data specification of "name". Such an update is repeated until all conditions are processed and the final node is reached.

2.2.3 Generation of the levels

Several (finite) concrete values selected from the domain, which will be set to a factor, are assumed as the levels. To extract these levels, we basically apply the techniques of equivalent class division and boundary value analysis. To achieve adequate input space coverage, we extract not only the values which are inside

of the domain as normal levels but also the values which are outside of the domain as abnormal levels.

As for object data-type, if every attribute is “normal”, then the whole object is assumed to be a normal level. Otherwise, if any of the attributes is “abnormal”, then the whole object is assumed to be an abnormal level. In other words, after expanding an object into a tree structure, even if just one “abnormal level” exists as a “leaf”, the entire object is assumed to be abnormal.

2.2.4 Combination of levels

The test data is composed of the input and the expected result.

In general, the input is a combination of the levels set to factors. Since testing all combinations is unrealistic, a variety of techniques can be used to make the number more practical. Here, we adopt a simple goal, “level coverage”, to generate inputs which ensure that all levels generated are covered. This means, each level must used at least one time somewhere in the test data. In our method, the combinations that consist of all normal levels are generated, and these are assumed to be normal inputs. Because we generate the minimum number of normal inputs that can satisfy level coverage, the number of normal inputs equals the number of normal levels of the factor that has the most normal levels. On the other hand, combinations that include just one abnormal level are also generated, and these are assumed to be abnormal inputs. These tests aim to confirm that an abnormal input CANNOT trace the execution path. The inclusion of just one abnormal level makes fault localization easier when failure occurs.

The expected result corresponding to an input is acquired from the post condition associated with the execution path in the UML activity diagram.

3. Evaluation

3.1 Tool prototype

To verify and evaluate the method proposed above, we implemented a prototype of our “test design support tool”. A software design model based on UML 2.0 in XMI 2.1 format (a xml file) was assumed as the tool's input, and the output is a list of test cases and the test data used by each test case (csv files). Each test data consists of an ID, input values, expected result, and a boolean value which shows whether it's normal for the corresponding path.

3.2 Viewpoints

We evaluated whether the proposed method, intended to automate test design, is effective enough to secure the desired test density and the SUT coverage with minimal manual effort.

First we create and evaluate a UML model of an application, and then compare the test design artifacts made from the same UML model that are created by hand (manual) and by our generation method (automatic).

The viewpoints of this evaluation are shown below:

(1) Comparison of the manpower needed for test case extraction and test data creation.

(2) Comparison of the test density and the SUT coverage of those test cases (automatic and manual) that match, i.e. the test cases perform the same test.

3.3 Results

We used a Java web application, a shopping store, to carry out the evaluation. We used a server side order processing component of the application; its scale is about 4.8 KLOC.

To compare manual to automatic, the number of test cases is shown in Table 1. The matching test cases may have an “n to 1” relation, because we considered test cases “match” when they do the same confirmation, even seem not exactly same.

Table 1 Comparison of the number of test cases

	Manual	Automatic
Matching test cases	107 (56.3%)	360 (6.0%)
Non-matching test cases	83 (43.7%)	5597 (94.0%)
total	190	5957

The test density is shown in Table 2. In Table 2, “Automatic” is calculated using the number of matching automatic test cases (360 in Table 1).

Table 2 Comparison of test density

	Manual	Automatic
test density (number of test cases / KLOC)	39.6	75.0

The structure coverage of test cases and input space coverage of the test data are shown in Table 3.

Table 3 Comparison of SUT coverage

		Manual	Automatic
structure coverage		covered all execution paths	covered all execution paths
input space coverage	levels	representative values only	covered all boundary values
	combinations	without clear rules	covered all levels

We used one average skilled engineer for this evaluation and the manpower cost total is shown in Table 4. The unit man-power costs per test case and per scale are also shown in Table 4. Note that our prototype generates test cases within one minute so we did not count it in Table 4. The overhead cost of automatic includes the cost of retouching existing “roughly sketched” UML models to make them sufficiently detailed to generate test cases, this assumed to be 50% of the overall cost of UML modeling. In test execution after test design, the cost of test case selection will also be an overhead. Our research to date has not considered this process, but automation appears possible [9].

Table 4 Comparison of the manpower

	Manual	Automatic
for test design (man-minutes)	2330	0
for other overheads (man-minutes)	0	1123
total (man-minutes)	2330	1123
manpower per test case (man-minute / test case)	12.3	3.1
manpower per scale (man-hour / KLOC)	8.1	3.9

The above results confirm that the proposed method can reduce, for 56.3% of the manual test design work, 52% of the manpower. Moreover, the test density and the SUT coverage are greatly improved.

3.4 Discussion

The proposed method improves the test density and the SUT coverage with less manpower. However, some problems remain, as the method could not reproduce all manually made test cases, and the number of generated test cases is too large to permit practical test execution. We give a more detailed discussion of

the evaluation results below. We separate the matching test cases, i.e. the proposed method is successful, and those test cases without matching.

3.4.1 About matching test cases

These are the test cases created both by the human engineer and our tool. Most tests of the normal inputs to the execution paths belong to this category. Our tool improved the test density, but it needs to be enhanced through the introduction of orthogonal tables or pair-wise methods to narrow the number of combinations. With regard to structure coverage, all execution paths were covered by the automatic tool and by manual operation. Several differences were observed in terms of the input space coverage as follows:

(1) Extraction of levels

Our method tended extracted more levels and thus better coverage. For instance, when the constraint "arbitrary character string" is specified, only one test data was created by the engineer, while the automatically generated test data contained two or more variations based on boundary value analysis, i.e. different character string lengths.

(2) Extraction of Combinations

The manual process yielded relatively few combinations of factors (variables), while our tool covered all generated levels (values). For cost reasons, the expert focused on just the "key" factors influencing the branching of the execution paths. Given that our goal is comprehensive confirmation that software works correctly according to its design, we can say that the automatically generated test data is superior to the manually created test data and provides far better coverage.

3.4.2 Test cases created only by the expert

43.7% of the test cases created manually were not generated by the automatic tool. The typical reasons are listed below:

(a) A part of information is not described in the UML model

Generally, it is difficult to include information related to the semantics of the software in the UML model, and so the description is incomplete. For example, a variable may be defined as string type but numbers should be used to confirm successful operation, or various dependencies exist among two or more factors etc. It seems that it is possible to deal with this kind of knowledge by notation extension and setting the extension in the UML model. Another way is, if necessary information is formalized outside of the UML model, an extended method could be able to extract test cases considering both this information and the UML model.

(b) Two or more activities are processed in integrated manner.

Because experts are able to access all information in the UML model in a comprehensive manner, they can make test cases that considering internal branching in the sub-activities called by the test target activity. The current version of our proposed method pays attention only to one activity and does not consider the nested structure of activities. To mimic the abilities of experts, test case extraction based on integration testing should be realized.

3.4.3 Test cases created only by tool

Almost all abnormal inputs belong to this category. Most abnormal (error handling) tests created manually were just "null" tests. In contrast, in addition to the cases of "null" usage, our method generates all cases contain abnormal values at the "leaf level" somewhere in the tree structure of an object. These are output as abnormal test data. However, as well as the test by normal inputs, "many more" does not mean "better". It is necessary to narrow down the number of test cases with abnormal inputs to a practicable level. For example, for object data-type, we should identify and focus on only the important cases only while ensuring input space coverage.

4. Conclusions

We proposed a method that can automatically extract test cases and test data with hierarchical structure,

from UML design models. We create a tool prototype to evaluate the method.

Future works are the following. First, we confirmed the effective and ineffective points of the method by using a “toy application” that was relatively small. We need to conduct an application experiment on an ongoing project, to get more reliable evaluation results. Second, we need to gradually eliminate various flaws to improve the proposal. The main flaws include the following. (1) Cannot treat variables other than the input parameters of an activity, as factors. (2) Cannot handle conditions wherein two or more factors are dependent. (3) Cannot handle the case when the value of the factors are overwritten and changed during the execution path. The priority of the constraints will be clarified by an application experiment on a real project. Third, our test design automation proposal can easily generate a large number of test cases, but if it takes a great deal of manpower to conduct all the tests generated, this is not significant advantage. A solution may be altering the output of our test design automation tool to support the use of test execution automation tools such as xUnit [10].

The proposed method can, for a significant part of the test design process, achieve higher SUT coverage and test density with fewer man-hours than is possible with manual creation. We intend to achieve more efficient software quality assurance by refining the work described herein and extending it. This paper considered only UML design models but other notations exist and it appears likely that our proposal could utilize these notations as its source of design information.

References

[1] Dias Neto, A.C., Subramanyan, R., and Vieira, M., Travassos, G.H., A survey on model-based testing approaches: a systematic review, *In WEASELTech '07: Proceedings of the 1st ACM international workshop on Empirical assessment of software engineering languages and technologies*, pp. 31-36, ACM, 2007.

[2] Hartman, A., AGEDIS-Model Based Test Generation Tools, *Agedis Consortium*, 2002.

[3] Prasanna, M., Sivanandam, S., Venkatesan, R., Sundarrajan, R., A survey on automatic test case generation, *Academic Open Internet Journal* 15, 2005.

[4] Edvardsson, J., A survey on automatic test data generation, *In Proceedings of the 2nd Conference on Computer Science and Engineering*, pp. 21-28, Linkoping, 1999.

[5] Korel, B., Automated software test data generation, *Software Engineering*, IEEE Transactions on 16(8), 870-879, Aug 1990.

[6] McMinn, P., Search-based software test data generation: a survey, *Software Testing, Verification & Reliability* 14(2), 105-156, 2004.

[7] Black Sheep Web Software, generatedata.com, http://www.generatedata.com

[8] IEEE, IEEE Standard Glossary of Software Engineering Terminology/IEEE Std 610.12-1990.

[9] Jussi Kasurinen, Ossi Taipale, and Kari Smolander, Test case selection and prioritization: risk-based or design-based?, *In Proceedings of the 2010 ACM-IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM '10)*, 2010.

[10] Gerard Meszaros, Xunit Test Patterns: Refactoring Test Code. Prentice Hall PTR, Upper Saddle River, NJ, USA. 2006.