View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Hosei University Repository

法政大学学術機関リポジトリ

HOSEI UNIVERSITY REPOSITORY

A Software Tool Support for “Vibration” Testing

著者	Zhao Pan
出版者	法政大学大学院情報科学研究科
journal or publication title	法政大学大学院紀要. 情報科学研究科編
volume	13
page range	1-6
year	2017-03-31
URL	http://doi.org/10.15002/00021515

A Software Tool Support for "Vibration" Testing

Pan Zhao
Graduate School of Computer and Information Sciences
Hosei University
3-7-2 Kajino-cho, Koganei-shi, Tokyo 184-8584, Japan
E-mail: pan.zhao.78@stu.hosei.ac.jp

***Abstract*—"Vibration" testing presents a strategy for generating test cases from an atomic predicate with the aim of achieving full coverage of the paths in the program that implements the function defined by the predicate in the specification. However, how to efficiently generate adequate test cases using the strategy from the same predicate to quickly traverse all of the related paths in the program is still an open problem. In this paper, we describe a prototype software tool we have built recently that supports the test case generation based on the principle of the "Vibration" testing and present an experiment to evaluate the effectiveness of the tool supported—"Vibration" testing.**

Keywords—specification; vibration; test case

I. INTRODUCTION

Test cases are essential for software testing, and how to use minimum manpower, resources, and time to generate adequate test cases to ensure software quality is a goal that many software companies pursue. Each software product and development project needs a set of effective test plans and methods. Software testing is an unstable work because there are many factors affecting software testing, such as the complexity of software, the ability of developers, and the execution of test methods. Because of this feature, testing work needs a set of appropriate test cases in order to improve the stability of software testing. Thus, regardless of whoever performing the software testing work, what he or she only needs to do is referring the test cases to implement this work. Minimizing the impacts of human factors, the efficiency of testing work can be guaranteed. Therefore, designing and generating a set of adequate test data is an important process.

Likewise, automatically generating test cases should be attached importance. The process of test case designing accounts for 60% of the whole testing process. If the test case designing work is done only by humans, it is not only inefficient, but also difficult to guarantee the effectiveness of the test cases. To solve this problem, there is a large body of research on specification-based testing [10]. Many projects have worked out techniques for automated test case generation from specifications, such as Z specifications [3], UML statecharts [4], or ADL specifications. But for these methods, they all need to be asked, whether they can detect all defects in a program. Obviously this is extremely difficult to realize. This situation has motivated us to carry out the research presented in this thesis.

In this thesis, our research focuses on how to automatically generate test cases from formal specifications to traverse as many representative paths in a program as possible by the "Vibration" method proposed by Liu in [1]. To realize the "Vibration" method, Liu has also developed an algorithm to implement the method. The basic idea of the algorithm is to repeatedly generate test cases from the same atomic predicate (a relation) by properly changing the "distance" between the two expressions in the atomic predicate. Our contributions in this thesis include:

- Developing a software tool to support the flexible use of the algorithm by freely adjusting the "distance" to generate test cases from a SOFL atomic predicate.
- Conducting experiments to evaluate the effectiveness of the tool supported "Vibration" method in terms of the relation between the number of the generated test cases and the path coverage in the program implementing the predicate.
- Applying the SOFL three-step formal specification approach to develop the supporting tool in order to ensure the reliability of the tool.

From the next section, we will first introduce the SOFL specification and the functional scenario-based test case generation method, and then discuss the essential idea of the "Vibration" method. After that, we will explain about the tool we have built and describe several experiments for the evaluation of the tool supported "Vibration" testing.

II. SOFL SPECIFICATION AND FUNCTIONAL SCENARIO-BASED TESTING

Formal methods for developing software are used for formal specifications and for program verification based on formal language [2]. SOFL(Structured Object-Oriented Formal Language) is one of the Formal engineering methods for industrial software development. As a model-based formal engineering method, it integrates the advantages of Data Flow Diagrams, Petri nets, and VDM-SL(Vienna Development Method-Specification Language) [2]. Furthermore, it extends the traditional Data Flow Diagram, and uses Petri net to provide operational semantics for DFD, which forms CDFD (Condition Data Flow Diagrams). The CDFD and its associated modules are organized in a hierarchy, so that decompose the complex process into a number of well organized sub-CDFDs, and then, VDM-SL is used in describing the components of the CDFD, known as Module Specification. In order to describe a

Supervisor: Shaoying Liu

software system, SOFL method consists of two parts: CDFD(Condition Data Flow Diagrams) shows the relationship between processes, and specification expresses the definition of each process. In this section, we just focus on the introduction of SOFL specification. An example is as follows:

process *A (x: int) y: int*
pre *x > 0*
post *x <= 10 and y = 1 / x or*
*x > 10 and y = x * 5*
end_process

As shown above, we can write a specification as $S(S_{iv}, S_{ov})[S_{pre}, S_{post}]$, where S_{iv} is the set of input variables and S_{ov} is the set of output variables, and the pre- and post- conditions define the functionality of the operation. In a previous research [5], a concept of functional scenario was proposed. A set of functional scenarios can be derived from the specification, each defining an independent function in terms of input - output relation. Let

$S_{post} = (G_1 \wedge D_1) \vee (G_2 \wedge D_2) \vee ... \vee (G_n \wedge D_n)$,

where G_i is a guard condition and D_i is a defining condition, and i = 1,…, n. Then, a functional scenario form (FSF) [5] of S is:

$(S_{pre} \wedge G_1 \wedge D_1) \vee (S_{pre} \wedge G_2 \wedge D_2) \vee ... \vee (S_{pre} \wedge G_n \wedge D_n)$,

where $S_{pre} \wedge G_i \wedge D_i$ is called functional scenario. The functional scenarios of the above example are the following three:

- $x > 0 \wedge x <= 10 \wedge y = 1 / x$
- $x > 0 \wedge x > 10 \wedge y= x * 5$
- $x <= 0 \wedge$ Anything

A scenario-based testing method has been presented in [8, 9], and this method indicates the functional independence of each functional scenario. The central idea of this testing method is: generate test case for each functional scenario at least once by satisfying each atomic predicate of pre-conditions and guard conditions ($S_{pre} \wedge G_i$) in a scenario. There are already some algorithms for the automatic test data generation based on an atomic predicate [8], but for these algorithms, they only can produce one test case each time, moreover, a functional scenario is usually refined into a collection of program paths and it is also extremely difficult to establish any theory that tells how test cases can be generated to ensure that all of the paths can be traversed. On the basis of the scenario-based testing, in order to produce sufficient test cases based on an atomic predicate, a "Vibration" testing method was put forward in [1].

III. Introduction to "Vibration" Testing Method

A. *Principle of "Vibration" Method*

The purpose of the "Vibration" method is to generate an appropriate test set automatically from a predicate relational expression, so that all of the representative paths in a program can be traversed at least once. In this part, we will introduce the principle of the method. Firstly, there is an atomic predicate that is a relation $E_1(x_1, x_2, \dots, x_n)\ R\ E_2(x_1, x_2, \dots, x_n)$ from a functional scenario, where R is a relational operator, such as > , < , >= , or <=, and E_1 and E_2 are expressions that contain all the input variables $x_1, x_2, \dots, x_n$. Secondly, if the input variables are numerical type, there is a "distance" between E_1 and E_2 defined as the absolute value of E_1-E_2. The distance vibrates (change repeatedly) in a regular way between the initial value and the maximum value. Finally, according to the vibrating distance, repeatedly produce the values for all input variables from the relational expression, so that the test cases are able to be generated automatically.

B. *Test Case Generation*

After a brief introduction to the principle of the "Vibration" method, we will discuss how it works specifically. In this paper, two important points are mentioned above: an atomic predicate and the "distance" between the two expressions in the relation. The "Vibration" method is mainly used to generate test cases by the "distance" vibration. First of all, we have a relation $E_1(x_1, x_2, \dots, x_n)\ R\ E_2(x_1, x_2, \dots, x_n)$ (e.g. , x + 5 > y, where "x + 5" is like $E_1(x_1, x_2, \dots, x_n)$, ">" is like R, and "y" is like $E_2(x_1, x_2, \dots, x_n)$) which contains all input variables from a functional scenario (e.g. $x + 5 > y \wedge z = x + y$, where pre-condition is "true", guard condition is "x + 5 > y", and defining condition is "z = x + y") in SOFL specification. This scenario is the basis for generating test cases. To begin with, the user should defines an initial value for distance (e.g., distance = 8) , as well for the range of distance increase and decrease and the number of test cases, increasing value should be greater than the reducing value, write as distance_up and distance_down, (e.g., distance_up=8, distance_down=5). These initial variables should be appropriate based on different applications. Next, according to the initial distance, the program first produces values randomly for all input variables, at the same time, two conditions must be satisfied that the relation between the two expressions still hold, and the absolute value of the two expressions is equal to the initial distance (e.g., x + 5 - y = 8, let, x = 5, y = 2) . In this way, the first test data is generated. And then, the distance increases by adding distance_up, a new distance is obtained (e.g., distance = 8 + 8 = 16). Likewise, generate the corresponding condition satisfying test data for the new distance. And then distance decreases, the same as the previous steps to generate test data. All in all, the distance increases and decreases regularly, and each time a distance change is made, a test data is generated, until the end command is issued. Finally, a test set is generated, which test efficiency is proved as the number of being covered paths, and it depends on the value of the initial variables defined by users, in another word, it depends on the distance vibration.

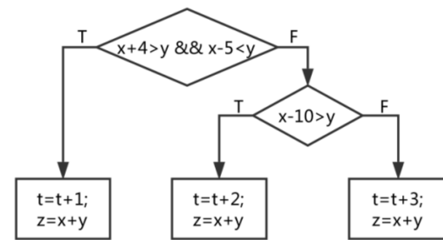

Fig. 1. Functional scenario paths

In the second section of this paper refers to that a functional scenario usually contains multiple paths, for traversing these paths as quickly as possible, the "Vibration" method is necessary and effective. Here we use the example that is

mentioned above to prove the necessity and efficiency of using "Vibration" method. The functional scenario is "x + 5 > y ∧ z = x + y", where "x" and "y" are input variables, "z" is output variable. In this scenario, there are three paths as showing in Fig.1. If the initial distance is 8, distance_up is 8, distance_down is 5, the relation is "x + 5 > y", then generate the test data: 1) x = 33, y = 30; 2) x = 66, y = 55; 3) x = -46, y = -52. After calculation, these test cases are able to cover all of the scenario paths. So using "Vibration" testing method, we can quickly cover the program paths and the efficiency of this method also can be guaranteed.

Obviously, using non-automatic "Vibration" method to generate test data by a person is complex and cumbersome, hence building a supporting tool to efficiently use this method is necessary.

IV. Supporting Tool for "Vibration" Method

A prototype tool has been built by us, called "Vibration" Testing Tool (VTT). Its main role is to support the V-Method, the user only needs to input the necessary initial values, click the button, and then the tool can automatically generate test cases by using vibration method. The algorithm for the "Vibration" method has been published in [1]. We have used Visual Studio 2015 to complete the development of the tool with C# language, and used SOFL three steps method to develop the tool.

A. *Functions*

Fig. 2 is the CDFD of the tool functions, which has been drawn using the SOFL Tool. As the CDFD shows, the tool performs four main functions:

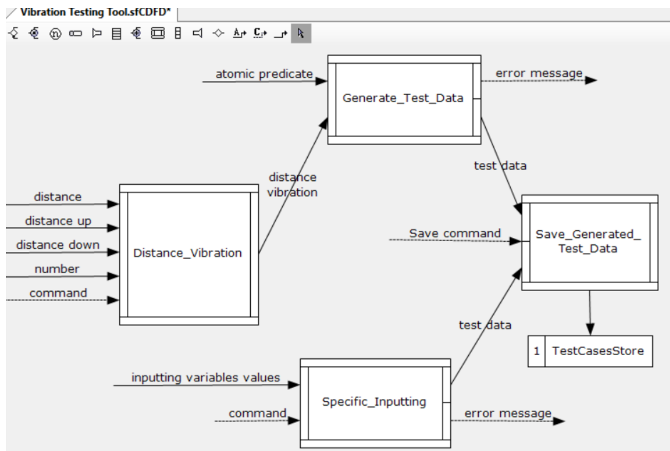

Fig. 2. CDFD of VTT

1) Produce "distance" values: Produce corresponding number of "distance" based on the relative input variables, which will be used for test data generation. The relative input variables are "Distance", "Distance up", "Distance down" and "Number". By the method of vibration, the "distance" increases or decreases regularly from the initial value. Finally, a set of "distance" values are generated, which number is determined by users.

2) Generate test data: In this process, except receiving the output results from the first process, there is an atomic predicate just like "ax + b > y" (linear relation), where "a" and "b" are constant parameters, "x" and "y" are input variables, is also received by this function. Under the condition that the atomic predicate is kept constant, based on the previously produced "distance", generating the requested number of test cases for each input variable. In addition, the incorrect input will cause that the tool displays a row of error message to remind users to pay attention to the correctness of the input values. There are two rules for input values: 1. The relation symbol must be selected from the drop-down menu; 2. The "if" statement in C# language for returning error messages is: if (d < 0 || up <= down || ((s == " < > " || s == " > " || s == " < ") && d == 0)) , where "d" is distance, "s" is the relation symbol.

3) Specific inputting: In the process of testing work, testers usually want to assign the specific test data what they want to test. Our VTT provides this function. Users can input the specific data they want to test based on the "distance" values they specify. In other words, the purpose of this function is that turn the automatically generation testing into artificial generation testing, meanwhile the characteristics of the V-Method are still retained. Fig.3 shows an example of the specific inputting function which were presented in Sect.III.B. Users input the "distance" satisfying values for each input variable and then the VTT can not only generate the group of test data one by one, but also check the availability of the input values.

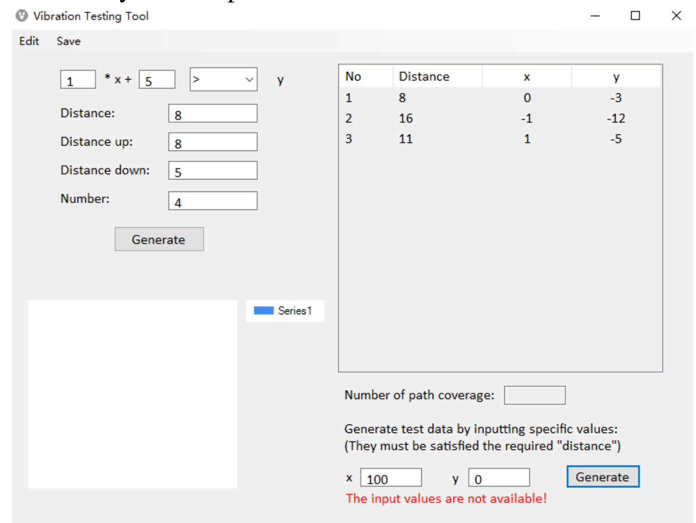

Fig. 3. Specific inputting function

4) Save the result: Save the generated test data to an excel file for checking in the future.

B. *Tool development process—SOFL three-step method*

When developers use SOFL to write specifications, this work consists of three steps: from informal specification to semi-formal specification, and then to formal specification [2].

1) Informal specification: In order to achieve a well organized informal specification, the following three aspects need to be considered as being presented in Fig.4.

2) Semi-formal specification: The overall architecture of the specification has been formed in the semi-formal specification, but some conditions still are described in natural language. A part of semi-formal is given below:

function *Distance_Vibration (distance: real, distance_up: real, distance_down: real, number: nat, command: bool) : seq of distance*
pre *The input values are available*
post *On the basis of the previous distance value, the operation is added or reduced to get a new distance value, and each distance value is stored in a sequence*
end_function

V.ifSpec

1 Functions
1.1 Produce corresponding number of "distance" values
1.2 Generate test data
1.2.1 Generate test cases from atomic predicate based on the first function results
1.2.2 Check the standardability of input values and show the error massages
1.3 Input the specific test data by users for inputting variables
1.4 Save the results to an Excel file
2 Data Resources
2.1 The input values are referenced by users from SOFL specification functional scenarios
3 Constraints
3.1 The relation symbol must be selected from the drop-down menu
3.2 distance >= 0
3.3 The vibrate trend of distance should be increase, that is "distance up > distance down"
3.4 Specific values for inputting variables must be satisfied the required "distance"

Fig. 4. Informal specification of VTT

3) Formal specification: It eliminates the natural language description of the document to provide a full SOFL formal document. A part of formal specification of VTT is:

function *Distance_Vibration (distance: real, distance_up: real, distance_down: real, number: nat, command: bool) : seq of distance*
pre *distance >= 0 and distance_up > distance_down*
post *if command = true then change_value = distance_up and command = false*
else change_value = -distance_down and command = true
*if distance = distance + number div 2 * distance_up - (number - 1) div 2 * distance_down /*The last "distance" value calculation formula.*/*
then Distance_Vibration = [distance]
else
Distance_Vibration=conc(distance,Distance_Vibration (distance+change_value, distance_up, distance_down, number, command)
end_function

V. Experiment

Three case studies and a comparison for the test results between "Vibration" method and "Pairwise testing" methods are given in this section. The reason we use "Pairwise testing" method for the comparison is that it is a popular technique for test case generation [6], often used in industry [7], and suitable for automation [1]. The following gives descriptions of these cases, which are all derived from our life. Using VTT to generate test data for them, comparing the test results with "Pairwise testing" method. The aim of this section is proving the practicability, feasibility, and effectiveness of "Vibration" method.

A. BMI (Body Mass Index) calculation

BMI is a commonly used measure standard for measuring the degree of obesity (5 levels) and whether a person is healthy or not, and it is a relatively objective parameter by body weight and stature. The formal specification of the BMI calculated process and the reference standard (divided into five paths) are:

process *adult_BMI_calculation(stature:real,weight: real) BMI: real*
pre *stature>140 and stature<220 and weight>35 and weight<150*
post *weight > 0. 63 * stature - 66.15 and BMI = weight / (stature * stature / 10000)*
end_process

TABLE I. BMI REFERENCE STANDARD

1	BMI < 18.5	Underweight
2	18.5 <= BMI < 23	Normal
3	23 <= BMI < 25	Overweight
4	25 <= BMI < 30	Obese
5	BMI >= 30	Clinically Obese

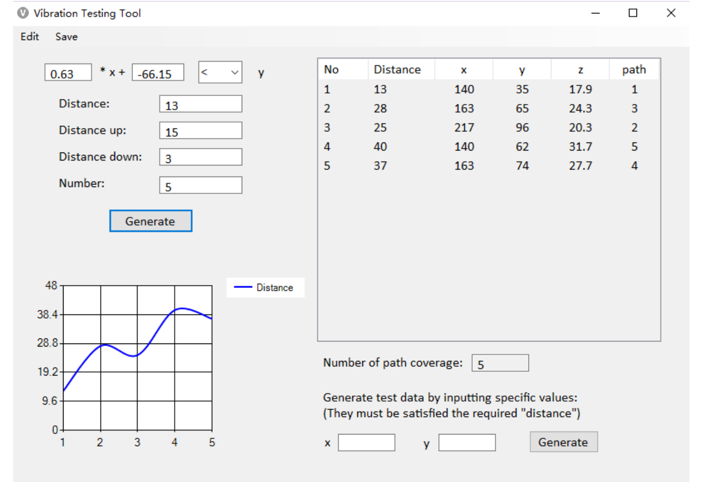

Fig. 5. Test data generation for BMI calculation

In this case, the guard condition "weight > 0. 63 * stature - 66. 15" is the atomic predicate which will be used in generating test cases. This relation is figured out by the body weight standard formula and actual situation, as a matter of fact, the weight of a human body cannot be below a certain value. Based on the pre- and post- conditions and after several tries, we infer the appropriate input variables respectively, which are: distance = 13, distance up = 15, distance down = 3, number = 5. Using these input values, VTT generates only five test cases, and covering all of the paths in this specification! That is to say, each vibration of "distance" makes the test case covering a new path. The UI of this case is shown in Fig. 5. The input variable "x" is stature, "y" is weight, and for this example, we add two variables "z" and "path" in the tool to show the results of the vibration method more intuitively, where "z" is the calculated

BMI, “path” visualize the traversed path number of the corresponding test case.

B. Balance ratio calculation

Balance ratio refers to the ratio of the balance and income of a family in a certain period. It reflects the ability and saving awareness of the family to control spending, and is the basis for future investment and financing. The SOFL specification is:

process *balance_ratio_calculation(income: real, expenditure: real) balance_ratio: real*
pre *income>1000 and income<20000 and expenditure>0*
post *income>=expenditure and balance_ratio = (income - expenditure) / income*
end_process

Income and expenditure are the input variables, balance ratio is output variable. Average dividing 10 parts from 0% to 100% for the only functional scenario. Setting the atomic predict relation is “income >= expenditure”. Then, make the same attempt as the first case to generate test data. Fig.6 shows the relevant data that generates the test case, where “x”, “y”, “z”, and “path” respectively are “income”, “expenditure”, “balance ratio” and “the covered path number”.

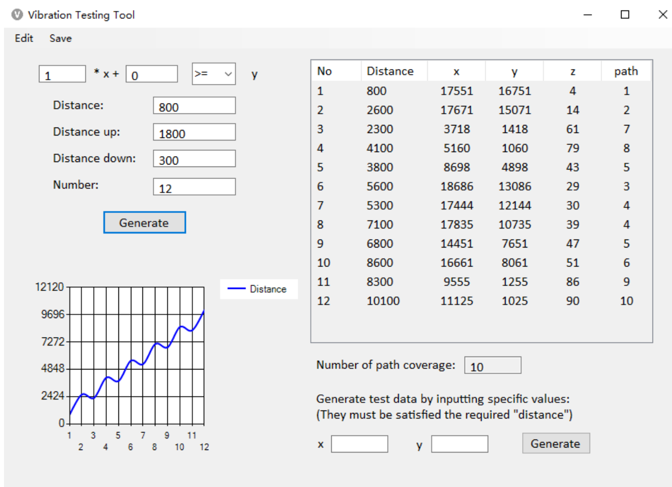

Fig. 6. Test data generation for Balance ratio calculation

C. Train fare system

Train fare system is used to calculate the cost for passengers based on their starting station and arrival station. In this case, we use the train which has the most number of stations (57 stations) in China. After calculation based on starting station number and arrival station number, it contains 1596 paths in this case. Here is the specification of the system:

module *train_fare_system*
type *starting = nat;*
arrival = nat;
fare = real;
Fare_schedule = map starting and arrival to fare;
process *train_fare_calculation (starting: real, arrival) fare: real*
ext *rd fare_schedule_file: Fare_schedule*
pre *starting < arrival*
post *arrival - starting <=56 and fare = fare_schedule_file (starting, arrival)*
end_process

Starting and arrival are the input variables and fare is output variable. In post condition, for veracity of this experiment, it actually be separated 5 functional scenarios. The reason of it is the shorter the distance between arrival and departure stations, the more the paths to be traversed. Thus for the veracity of the test, separating generate test data is much better. As above cases, Fig.7 presents the UI of using VTT to generate test cases, where “x” and “y” are input variables “starting” and “arrival”.

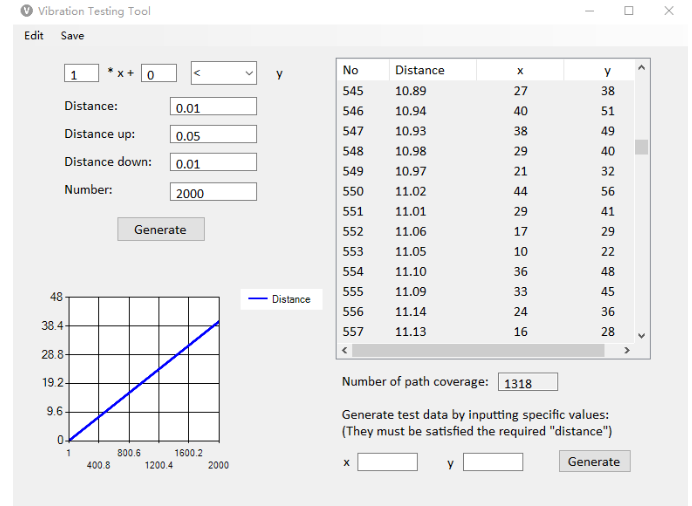

Fig. 7. Test data generation for train fare system

D. Experiment results and analysis

In line with the principle of covering all possible paths in a program, test data number should equal to or greater than path number of the program because each test data can only cover one path in the program. For above three experiments, the number of test data is determined by the minimum number that can make one of the testing methods path coverage rate reach up to 100%. But for large-scale program such as the third case, path coverage rate is almost impossible to reach 100%, therefore the select principle is that through several attempts to determine the number of test data which path coverage rate is the biggest. Because that if there is enough test data, no new paths will be traversed as the number of test data increases. Table.II lists the path coverage rate of each testing method for these three cases:

TABLE II. PATH COVERAGE RATE RESULTS

Programs (number of paths, number of test cases)	Pairwise testing	Vibration testing
BMI calculation (5, 5)	80%	100%
Balance ratio calculation (10, 12)	50%	100%
Train toll system (1596, 2000)	61%	83%

Table.II shows the results of the case studies. For the first case, the path coverage rates by using “Pairwise testing” and “Vibration testing” respectively are 80% and 100%, and for the

second case, the path coverage rates are 50% and 100%. This two cases both have fewer paths. The third is a large-scale case, the path coverage rates are 62% and 83% respectively. The table reflects 3 features: 1)The efficiency of test data coverage path generated by "Vibration" method is better than that by "P" method; 2)For small paths scale programs, the "Vibration" method can traverse all paths in the case of generating as few test cases as possible (even the same number as program paths). 3)For big scale program like the third case, the "Vibration" method does not guarantee that all paths can be covered. In the second case, we generate 3*4 numbers test cases, that is two input variables are produced 3 values and 4 values respectively in their domain, and they are evenly distributed in the domain, and then pairwise combine. Since there is a lack of definition of the relation between input variables in the second case, "Pairwise testing" producing test cases always ineffective to test the program, which can reflect the limitations of this method.

The experimental results show that the "Vibration" method generates more adequate test data through the specific predicates confirmation and several attempts for "distance" vibration, which makes the generated test cases covering paths in program as much as possible, meanwhile, all of the input variables should satisfy the constraints of the program. Besides, the distance vibration makes test data more randomness and more wide coverage. Actually, using "P" method usually generate some ineffective test data because it cannot define the relation between the input variables, in this case, the "Vibration" method shows its superiority.

VI. Discussion and Conclusion

The decision of "distance" is a discussing worthy problem in the use of "Vibration" method. If the under test program contains a lot of paths, in another words, the distribution of program paths is quite dense, then the vibration of "distance" should be more frequent and its amplitude should be smaller, so as to ensure that more paths can be covered. On the contrary, if the paths distribution in program is dispersed, that is the required distance between input variables is large, then the "distance" vibration amplitude should be bigger and frequency should be lower, so that avoid that too many test cases are generated to cover the same paths. Therefore, the "distance" needs to be decided according to different programs. The best testing status is, every time the distance changes, a new path is covered by the generated test case, under this premise, try to reduce the vibration amplitude, to find whether there are some haven't covered paths or not because of the wide vibration of "distance", until the number of being able to be traversed paths reach to maximum by a set of "distance" related values ("distance up" and "distance down"). The VTT requires users to attempt several times to decide the most appropriate "distance", thus users need to be familiar with the test object, as well as have the abilities of observation and analysis.

A small discussion about "Pairwise testing" should be mentioned. In Sect.5.4, we have mentioned the limitations of the "P" method in the second case. Actually, if we use 1*12 numbers test cases to pairwise test, the path coverage rate will be 100%. But it's only a special circumstance and can't represent normal situations. Thus the limitations still remain.

In this paper, we have described the VTT for supporting the "Vibration" testing method. The using of this tool mainly consists of three steps: 1)determining the predicate relation from specification scenario; 2)deciding the appropriate initial values (distance, distance up, distance down, number) based on the under test program; 3)automatically generating test data. In addition, the effectiveness of the method is proved by several experiments, and it also can be widely used in various fields.

VII. Future work

At present, VTT is a simple prototype tool that only provides functions to support the "Vibration" method. The types of the input variables are also limited to numeric types. In previous studies, the definitions of "distance" for other data types such as set types, char types, and enumeration types are mentioned [1], but none of them has been implemented by a tool. Thus, one of the future works for "Vibration" method is to improve the tool that can support more data types. Second, the definition of "distance" is what we attach importance to. It seems unreliable and inconvenient that decide the "distance" only by users, hence we propose a function for the VTT that supports recommending the value of "distance" for users based on the previous paths traversal situation. Third, in this tool, only linear relations can be used in atomic predicates, for more extensive use, the future work should focus on the other types of atomic predicates.

References

[1] Liu, S., Nakajima, S.: A "Vibration" Method for Automatically Generating Test Cases Based on Formal Specifications. In: Software Engineering Conference, 2011 18th Asia Pacific, pp. 73-80, 5-8 Dec. 2011.

[2] Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. In: Springer, Heidelberg (2004).

[3] Horcher, H.-M.: Improving software tests using Z specifications. In: Proceeding 9th International Conference of Z Users, The Z Formal Specification Notation (1995).

[4] Offutt, J., Abdurazik, A.: Generating tests from uml specifications. In: France, R. B.(ed.) UML 1999. LNCS, vol. 1723, pp. 416-429. Springer, Heidelberg (1999).

[5] Liu, S., Nagoya, Y., Chen, M., Goya, McDermid, J. A.: An Automated Approach to Specification-Based Program Inspection. In: Lau, K.K., Banach, R., editors, Proceedings of 7th International Conference on Formal Engineering Methods(ICFEM2005), pp. 421-434, Manchester, UK, 1-4 Nov. 2005. LNCS 3785, Springer-Verlag.

[6] Tai, K. C., Lei, Y.: A Test Generation Strategy for Pairwise Testing IEEE Transactions on Software Engineering, 28(1), January 2002.

[7] Czerwonka, J.: Pairwise Testing in Real World. In: proceedings of 24th pacific Northwest Software Quality Conference, 2006.

[8] Liu, S., Nakajima, S.: A Decompositional Approach to Automatic Test Cases Generation Based on Formal Specification. In: 4th IEEE International Conference on Secure Software Integration and Reliability Improvement, Singapore, pp. 147-155, IEEE CS Press, 9-11 June 2010.

[9] Li,M., Liu, S.: Automated functional scenario- based formal specification animation. In: 19th Asia-Pacific Software Engineering Conference, PP. 107-115. IEEE CS Press (2012).

[10] Donat, M.: Automating Formal Specification-Based Testing. In: Proceedings of TAPSOFT' 97, pp. 833-848, Lille, France, April 1997. Springer-Verlag