Research on the Vibration Testing Method and a Tool to Support it

Dissertation submitted in partial fulfillment for the
degree of Master of Engineering

Kenya Saiki

Under the supervision of
Professor Shaoying Liu

Dependable Systems Laboratory,
Informatics and Data Science Program,
Graduate School of Advanced Science and Engineering,
Hiroshima University, Higashi-Hiroshima, Japan

February 2023

Abstract

The Vibration testing method (V-Method) was proposed and developed to deal with the problem of how to generate test cases based on formal specifications to achieve the coverage of representative program paths in recent years, but it lacks a tool to support it. In this paper, we present a software tool to support the V-Method by elaborating on the algorithms for automatically generating test cases based on various kinds of logical expressions and data types, evaluating test results, and managing documents in relation to test cases. We also present some experiments to evaluate the performance of our tool. Finally, we conclude the paper and point out the future research direction.

Acknowledgements

I would like to extend my sincere gratitude to Professor Shaoying Liu, the supervisor of my study, for his constant guidance, kind advice and continuous encouragement throughout the progress of this work. I wish to thank Professor Tadashi Dohi, Professor Hiroyuki Okamura and Professor Toru Nakanishi for their many invaluable comments, constant support, helpful guidance and warm encouragement. Finally, it is my special pleasure to acknowledge the hospitality and encouragement of the past and present members of the Dependable Systems Laboratory, Informatics and Data Science Program, Graduate School of Advanced Science and Engineering, Hiroshima University.

Contents

Chapter 1

Introduction

Formal methods are one of the techniques for software development[1]. It is known as a method for developing reliable software by mathematically rigorous descriptions[1]. Its advantages include the potential for automatic test case generation and test results analysis but face a challenge in generating test cases to achieve path coverage in programs.

The Vibration testing method has been put forward by Liu [2] to address this challenge. It focuses on the difference as "distance" between the left side and the right side of an atomic predicate used in formal specifications, and on changes of the distance in order to obtain effective test values. Generating test cases by vibrating the distance in this way yields test cases that are expected to traverse as many paths of the program as possible.

However, this approach has several deficiencies. One is that when the test condition is a conjunction of atomic predicates, it is difficult to generate valid test cases because the Vibration method is designed to generate test cases only from one atomic predicate. Secondly, the Vibration method has not been sufficiently discussed to generate test cases from non-numeric data types. Finally, there is a lack of a matured tool support for the Vibration method.

We make three major contributions in this paper. Firstly, we make an extension of the Vibration Method. Specifically, we provide a new technique for setting the distance for the Vibration method based on conjunctions of atomic predicates. This will allow the Vibration method to be applied to conjunctions in test case generation. Furthermore, we have defined distances, for set types, sequence types, and composite types. It allows test cases to be automatically generated from these data types as well. Secondly, we overcame the deficiency of the current Vibration method not being able to generate test cases from non-linear expressions by proposing a way to apply the bisection method to automatically generate test cases from non-linear expressions. Finally, we have developed a supporting tool for the Vibration method with major functions: automatic test case generation, automatic test result analysis, and automatic theorem proving.

To evaluate the performance of the improved Vibration method and the developed tool, we have conducted several small experiments. The results show that our supporting tool provides sufficient and efficient support for test case generation, test result analysis, and theorem proving by testing.

The formal specification description language used in this study is SOFL, standing for Structured Object-Oriented Formal Language, and it was developed by Liu[3]. After a constant development of the SOFL specification language, the related techniques for developing software have been established. Now SOFL often refers to three main aspects: language, method, and processes. As a language, SOFL specifications support the effective integration of graphical and formal notations for constructability, comprehensibility, and maintainability. As a method, SOFL specifications adopt a three-step approach to constructing formal specifications and specification-based inspection and testing for verification and validation. It also combines the structured method and object-oriented method for software development. As a process, SOFL specifications adopt both evolution and refinement, and emphasize the paradigm of the first specification and then incremental implementation.

The remainder of this paper is organized as follows. Section 2 explains the Vibration testing method. Section 3 discusses our improvements to the Vibration method. Section 4 introduces the tool to support the Vibration method that we have developed. Section 5 shows the result of the evaluation of our work, the improvement of the Vibration method, and the supporting tool. Section 6 describes related work. Finally, in section 7 we conclude this paper and point out future research directions.

Chapter 2

The Vibration Testing Method

In this section, we introduce the traditional Vibration method, including the motivation, principle, and the remaining problems.

The Vibration testing method was proposed to overcome the disadvantage of formal specification-based testing. In formal specification-based testing, it is difficult to generate test cases that can ensure the desired path coverage because the program structure is treated as a black box. However, this deficiency can be improved by the Vibration method.

In the Vibration Method, we focus on the difference in the distance between the left side and the right side of a relation (i.e., atomic predicate) derived from the formal specification. Let $E_1(x_1, x_2, ..., x_n)\ R\ E_2(x_1, x_2, ..., x_n)$ denote that expressions E_1 and E_2 have relation R, where $x_1, x_2, ..., x_n$ are all input variables involved in these expressions. Then, let $|E_1 - E_2|$ be defined as the distance, denoted by d. Holding this relation, $E_1\ R\ E_2$, d can repeatedly change, which is like the vibration of some physical object. At a certain point of d, we can get an equation, $|E_1(x_1, x_2, ..., x_n) - E_2(x_1, x_2, ..., x_n)| = d$. A solution of this equation, which provides values for $x_1, x_2, ..., x_n$, respectively, will be treated as a test value.
For instance, given a relationship $x > 10$, and the distance is $x - 10$. When let the distance be 5, we can get an equation $x - 10 = 5$ and the test value $x = 15$ can be obtained. Then, repeatedly changing the distance to $1, 10, 3$, and 15, respectively we get $x = 11, 20, 13$, and 25 which constitute a test set. This is the basic principle of the Vibration method.

However, the Vibration method still has limitations. One is that the published work on the Vibration method so far has not shown how it works for non-linear expressions involved in an atomic predicate. Furthermore, the vibration method basically only supports numeric types and has not been adequately studied for non-numeric types. Another problem is that it is only well-defined for atomic predicates (i.e., relations), not for conjunctions of atomic predicates. In other words, how the Vibration method should work for conjunction is unknown. To efficiently use the Vibration method, a supporting tool is necessary, but it is not developed yet. In our work presented in this paper, these deficiencies are tackled and the corresponding solutions are provided.

Chapter 3

Improvements of the Vibration Testing Method

In this section, we describe how the Vibration Method is improved.

3.1 Automatic Test Case Generation from Non-Linear Expressions

The existing Vibration method generates test cases automatically by algebraic calculations. Therefore, test cases can only be generated from linear expressions. In this study, we introduce a numerical computation called the bisection method as a calculation method to address this problem. The bisection method is a method for calculating the solution to the equation numerically. The solution to the equation is found by repeatedly halving the width of the solution interval. This enables us to generate test cases even from non-linear expressions.

Algorithm 1 shows how the bisection method works. For example, let $x^2 - 3$ be $f(x)$ to find the solution of $x^2 - 3 = 0$. Let the initial values of x_0 and x_1 be 1 and 2 respectively. And also, let the tolerable error e be 0.1. In the first loop of the while statement in the algorithm, x_2 is assigned 1.5, and $f(1) * f(1.5)$ is positive so x_0 is assigned 1.5. In the second loop of the while statement, x_2 is assigned 1.75, and $f(1.5) * f(1.75)$ is negative so x_1 is assigned 1.75. In the third loop of the while statement, x_2 is assigned 1.625, and $f(1.5) * f(1.625)$ is positive so x_0 is assigned 1.625. In the fourth loop of the while statement, x_2 is assigned 1.6875, and $f(1.625) * f(1.6875)$ is positive so x_0 is assigned 1.6875. Here, $x_1 - x_0, 0.0625$, is less than e, 0.1, so we can get the value of 1.6875 as the solution of $f(x) = 0$. The changes of variable values in solving the equation $x^2 - 3$ are illustrated in Figure 3.1. In this example, we have set the tolerable error e to 0.1, but it is possible to make e smaller to obtain a more accurate solution.

3.2 Test Case Generation from Conjunctions

In the existing Vibration method, we can only generate test cases from an atomic predicate. This sometimes causes inefficiency. When we generate test cases from a test condition consisting of the conjunction of several atomic predicates, we can only focus on one atomic predicate each time, therefore we cannot generate efficient test cases.

We deal with this problem by classifying all conjunction into three types. Firstly, the conjunction involves only one variable. This type of conjunction can be expressed as $a < x < b$ by calculating the common range of the conjunction. For instance, let $5 < x \wedge x < 20$ be the conjunction involving only x. Then, we can derive an equivalent $5 < x < 20$. Then we set the distance to a certain value within the indicated range and apply the

Algorithm 1 The Bisection Method

```
Require: initial value: $x_0 < x_1$ and $f(x_0) * f(x_1) < 0$ and tolerable error: $e > 0$
Ensure: $x$: the solution of $f(x) = 0$
  while $x_1 - x_0 > e$ do
    $x_2 \leftarrow (x_0 + x_1)/2$
    if $f(x_0) * f(x_2) < 0$ then
      $x_1 \leftarrow x_2$
    else
      $x_0 \leftarrow x_2$
    end if
  end while
  return$x_2$
```

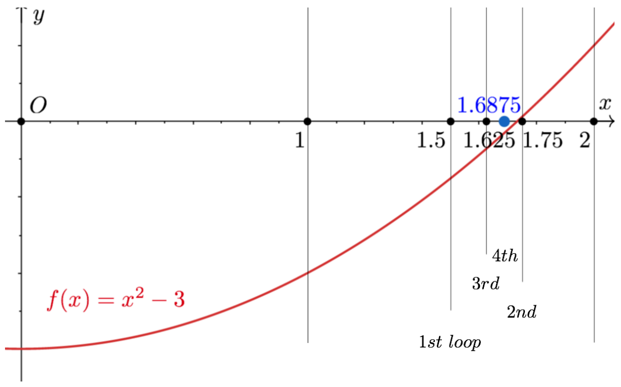


Figure 3.1: Changes of variables in solving the equation $x^2 - 3 = 0$ using the Bisection Method

Vibration method. Let $E_1(x)\ R_1\ E_2(x)\ \wedge E_3(x)\ R_2\ E_4(x)\ \wedge\ ,...,\ \wedge E_{2n-1}(x)\ R_n\ E_{2n}(x)$ be a conjunction consisting of n atomic predicates where E_i is a linear expression, x_i is an input variable, R_i is a relational operator, and $i = 1, ..., n$. This conjunction is then calculated and rearranged, we define the non-negative distance for it, as shown in Table 3.1. For example, when dealing with conjunction, $x \geq 1 \wedge x > 3 \wedge x \geq 5 \wedge x < 7$, this conjunction can be rearranged to $5 \leq x < 7$. This type of conjunction is the case in the third column in Table 3.1, so the distance is defined as $x - 5$, where $x - 5$ is smaller than 2. We can obtain the distance of this conjunction, then the distance is set to a real number, for example 1, this must be smaller than 2. Then we have an equation, $x - 5 = 1$. We can get a test case $x = 6$ by solving this equation.

Secondly, the conjunction involves more than two variables but each of the atomic predicates in the conjunction contains only one variable. For example, $x > 3 \wedge x < 5 \wedge y > 9$ is such a conjunction. Test cases of this type are generated by processing each variable individually. In this example, we consider $x > 3 \wedge x < 5$ and $y > 9$ as two test conditions in generating a test case. For $x > 3 \wedge x < 5$, the first type of idea is used to generate a value for x. For $y > 9$, this is an atomic predicate so we generate a value for y based on the conventional vibration method. Put the value for x and the one for y together, we will form a test case.

Thirdly, the conjunction involves two or more variables that are dependent on each other. For example, $x + y > 3 \wedge x + 2 * y < 4 \wedge x < 0$ is such a conjunction. Dealing with this type of conjunction with the Vibration method is difficult because there is no available technical solution for the problem of how to derive a common interval from the conjunction to facilitate the selection of the values for both x and y. In this case, we take a primitive approach that may be effective in some cases. In our approach, we focus on a single atomic predicate of the conjunction and apply the Vibration method to it to generate test cases. This idea is reflected

Algorithm 2 Test Case Generation from Conjunction

```
Require: number of repetition: $r > 0$
Ensure: $\boldsymbol{x}$: test case satisfying conjunction, $A_1 \wedge, ..., \wedge A_n$
    subscript of the atomic predicate: $i \leftarrow 1$
    counter: $c \leftarrow 1$
    test case: $\boldsymbol{x} \leftarrow$ GenerateTestCase($A_i$)
    while $\boldsymbol{x}$ does not satisfy conjunction and $i \leq n$ do
        while $\boldsymbol{x}$ does not satisfy conjunction and $c < r$ do
            test case $\mathcal{X} \leftarrow$ GenerateTestCase($A_i$)
            $c \leftarrow c + 1$
        end while
        $c \leftarrow 0$
        $i \leftarrow i + 1$
    end while
    if $\boldsymbol{x}$ satisfies conjunction then
        return$\boldsymbol{x}$
    else
        cannot generate test case
    end if
```

in Algorithm 2. Let $A_1 \wedge A_2 \wedge, ..., A_n$ be conjunction consisting of n atomic predicates where A_i is an atomic predicate. First, we generate a temporary test case from an atomic predicate, A_1. This temporary test case is then assigned into the remaining $n - 1$ atomic predicates to check that all the atomic predicates are true. If all atomic predicates are true, the temporary test case is accepted as a qualified test case. If not, we go back to the first step and regenerate a temporary test case. This is repeated several times until a true test case is generated. After several iterations, if no qualified test case can be generated, a temporary test case is generated from the atomic predicate, A_2. This process is repeated several times to generate a test case from this type of conjunction. This is still an idea and has not been implemented in the support tools presented in the next chapter.

Table 3.1: The Definition of the Distance for Conjunction

	rearrangement	distance	constraint
1	$a < x < b$	$x - a$	$0 <$distance$< b - a$
2	$a \leq x \leq b$	$x - a$	$0 \leq$distance$\leq b - a$
3	$a \leq x < b$	$x - a$	$0 \leq$distance$< b - a$
4	$a < x \leq b$	$x - a$	$0 <$distance$\leq b - a$
5	$a < x$	$x - a$	distance$\neq 0$
6	$a \leq x$	$x - a$	
7	$x < b$	$b - x$	distance$\neq 0$
8	$x \leq b$	$b - x$	
9	$x = a$	$x - a$	distance$= 0$
10	no solution	undefinable	

a and b are real numbers and $a < b$

3.3 Test Case Generation for Compound Types

Many formal specifications in SOFL use compound data types, such as set types, sequence types, and composite types, to improve their expressive power[3]. This will be likely to make logical expressions involve the operators defined in the compound data types. How to generate test cases from such a logical expression is still an open problem. In this section, we will discuss this issue on set types, sequence types, and composite types, respectively.

3.3.1 Set types

The set types are usually used for the abstraction of data items that have a collection of elements. A set is an unordered collection of distinct objects where each object is known as an element of the set. A set of values is enclosed with braces. For example, "$\{5, 9, 10\}$."

Our discussions in this subsection only focus on sets of numeric values and the same principle can be easily extended to sets of other types of values.

3.3.1.1 Distance Defined on Set Types

In order to generate test cases using the Vibration method, the distance between two expressions in an atomic predicate involving operators defined on set types needs to be defined. In this section, we discuss how we define such a distance and we will provide two strategy to define the distance for set types.

The first one is that the distance for test condition $E_1 <> E_2$ was defined as $|card(E_1) - card(E_2)|$ [2], where *card* means cardinality and the number of elements of the set. For example, for the test condition $\{1, 2, 3\} <> \{0, 2, 4, 6, 8\}$, distance is 2 since the cardinality of $\{1, 2, 3\}$ is 3 and the cardinality of $\{0, 2, 4, 6, 8\}$ is 5 and the differene is 3.

The second one is that the distance is defined as $\{d_1, d_2, ..., d_n\}$ for the test condition $E_1 <> E_2$, in addition to $|card(E_1) - card(E_2)|$. Where n is the cardinality with the greater cardinality in E_1 and E_2 and d_i is $e_{1_i} - e_{2_i}$, however when i is greater than $card(E_1)$ or $card(E_2)$ then d_i is just $-e_{2_i}$ or e_{1_i}, where e_{1_i} is the i-th element in E_1. Strictly speaking, this distance is not a set type in SOFL specifications. It is just expressed as a set type for easy handling. That is, in a set type, the values of the elements must not be duplicated, but the values of the elements of this distance can be duplicated. Besides, $|card(E_1) - card(E_2)|$ is redefined as $card(E_1) - card(E_2)$, as the previous one is unclear which set is longer, E_1 or E_2, if it is an absolute value. In this way, by giving the distance and also the value of the element, it is possible to generate a clearer test case. The following two examples are provided. First, for the test condition $\{1, 2, 3\} <> \{0, 2, 4, 6, 8\}$, distance is $\{1, 0, -1, -6, -8\}$ and -2 with regard to the cardinality. Then, for the test conditions $\{0, 2, 4, 6, 8\} <> \{1, 2, 3\}$, distance is $\{-1, 0, 1, 6, 8\}$ and 2 with regard to the cardinality. However, this idea is in fact insufficient Although it was the distance that determined the value of the elements, the order of the elements is completely irrelevant in the set type. In other words, distances that include this difference between elements may also be meaningless. However, the meaning for the first and second distance is the same, as there is no other way to give the distance randomly in the current vibration method. A second method is proposed for the future, when a more efficient way of giving distances is found out. On the other hand, the redefined distance for the cardinality will be better, because, we can find out easily which set is greater or smaller. Actually, on the vibration method, the distance is basically a positive number, but its redefine will be helpful for the unequal operators.

3.3.1.2 Test Case Generation for Set types

In this subsection, we describe how we generate test cases from set types using the second distance. We consider that all test conditions for set types can be divided into three categories. The first is a test condition that compares a variable S_1 of a set type with a variable S_2 of a set type, for example, $S_1 <> S_2$ (inequality) as well as $S_1 = S_2$. The second is a test condition comparing a variable S of set type with a set $\{e_1, e_2, ..., e_n\}$, where $e_i (i = 1, ..., n)$ is one of members of the set and may be a variable, for example, $S <> \{1, 2, 3, 4, 5\}$ or $S = \{a, b, 1\}$. Finally, the test condition compares a set $\{e_{1_1}, e_{1_2}, ..., e_{1_n}\}$ with a set $\{e_{2_1}, e_{2_2}, ..., e_{2_m}\}$, where e_{i_j} is one of members of the set and may be a variable, for example, $\{1, 2, 3\} <> \{1, 2, a\}$ or $\{2, 4, 6, 8\} = \{2, a, b, c\}$. We describe our approaches to test case generation for each of these categories.

For the first category,$S_1\ R\ S_2$, where R is the relational operator, we first give a random set to S_1, $\{e_{1_1}, e_{1_2}, ..., e_{1_n}\}$. Then we give the distance d with regard to the cardinality, where d is greater than or equal to $-n$ as the cardinality must be a non-negative number and $\{d_1, d_2, ..., d_m\}$, where if d is a positive number, m is n, else, m is $n - d$. The values of the elements of the set in S_2 are then calculated based on the definition of distance. The number of elements of S_2 is $n - d$. The i-th element of S_2 is $e_i + d_i$. However, if d is a negative number, that is, if the cardinality of S_2 is larger than the cardinality of S_1 and i is greater than n, the i-th element of S_2 is $-d_i$. Finally, if the elements of S_2 are duplicated, the duplicated elements must be removed. After the above process, the test case is generated.
For example, consider when S_1 is randomly given $\{1, 2, 3, 4, 5\}$ and distance for the cardinality is 3 and distance for elements is $\{2, 4, 3, 4, 5\}$, where the last three elements of the element distance, $3, 4$ and 5, match the last three elements of S_1, $3, 4 and 5$. At this time, another set type variable S_2 has a cardinality of 2 $(= card(E_1) - d)$ and the values of the element is $\{-1, -2\}$ $(= \{1 - 2, 2 - 4\})$. This means that the test cases, $S_1 = \{1, 2, 3, 4, 5\}$ and $S_2 = \{-1, 2\}$ are obtained.

Secondly, for the second category, $S\ R\ \{e_1, e_2, ..., e_n\}$, where R is the relational operator, we give the distance d with regard to the cardinality, where d is greater than or equal to $-n$ as the cardinality must be a non-negative number and $\{d_1, d_2, ..., d_m\}$, where if d is positive number, m is $n + d$, else, m is n, furthermore, if d is negative number, the last $|d|$ elements correspond to the last $|d|$ elements of $\{-e_1, -e_2, ..., -e_n\}$. At first, if the set $\{e_1, e_2, ..., e_n\}$ contains variables, give them random values. Then, the values of the elements of the set in S are then calculated based on the definition of distance. The i-th element of S is $e_i + d_i$. However, if d is a positive number and i is greater than n, the i-th element of S is d_i. Finally, if the elements of S are duplicated, the duplicated elements must be removed. After the above process, the test case is generated.
For instance, consider the test condition, $S <> \{a, 1, 2, 3, 4, 5\}$ and distance for the cardinality is -1 and distance for elements is $\{0, 2, 4, 6, 8, 5\}$. First of all, a random value of 10 is given for a. At this time, the values of the element of S are $\{10, 3, 6, 9, 12\}$. This means that the test cases, $a = 10$ and $S = \{10, 3, 6, 9, 12\}$ are obtained.
On the other hand, for the test condition, $\{e_1, e_2, ..., e_n\}R\ S$, where R is the relational operator, we give the distance d with regard to the cardinality, where d is smaller than or equal to n since the cardinality must be a non-negative number and $\{d_1, d_2, ..., d_m\}$ and if d is a positive number, m is n, else, m is $n - d$, furthermore, if d is positive, the last d elements must correspond to elements of $\{e_1, e_2, ..., e_n\}$. At first, if the set $\{e_1, e_2, ..., e_n\}$ contains variables, give them random values. Then, the values of the elements of the set in S are then calculated based on the definition of distance. The i-th element of S is $e_i - d_i$, however, if d is a negative number and i

is greater than n then the i-th element of S is $-d_i$. Finally, if the elements of S are duplicated, the duplicated elements must be removed, and this becomes a test case.

Finally, for the category, $\{e_{1_1}, e_{1_2}, ..., e_{1_n}\}$ R $\{e_{2_1}, e_{2_2}, ..., e_{2_m}\}$, where R is a relational operator, the distance for the cardinality is $n - m$. Then if m is greater than n, the distance, $\{e_{1_1} - e_{2_1}, e_{1_2} - e_{2_2}, ..., e_{1_n} - e_{2_n}, -e_{2_{n+1}}, ..., e_{2_m}\}$, is given, else, the distance, $\{e_{1_1} - e_{2_1}, e_{1_2} - e_{2_2}, ..., e_{1_m} - e_{2_m}, e_{1_{n+1}}, ..., e_{1_n}\}$, is given. The value of the element is known, so the value of the distance can be determined in this way, but when the i-th element in the set of test conditions is a variable, the value of the element of the set in the test condition is given randomly as d_i. Then, when e_{1_i} or e_{2_i} is variable, e_{1_i} or e_{2_i} is a test case, such as satisfying $e_{1_i} - e_{2_i} = d_i$, but if n is greater than m and i is also greater than m, e_{1_i} is d_i, else if m is greater than n and i is also greater than n, $-e_{2_i}$ is d_i. For instance, consider the test condition, $\{a, 1, 2\} <> \{2, 4, 6, b\}$ and distance for the cardinality is -1 and distance for elements is $\{a - 2, -3, -4, -b\}$. However, if the element contains a variable, the value of elements of distance is given, so the distance for elements is, for example, $\{3, -3, -4, 5\}$. Then, we can get equations, $a - 2 = 3$ and $-b = 5$, so $a = 5$ and $b = -5$. And, the test case $a = 5$ and $b = -5$ are obtained.

These are the test case generation methods for set types. Several operators are defined for set types in SOFL specifications. They are described in the following sub-sections.

3.3.1.3 Operators in Set types and Test Case Generation from them

In this subsection, we introduce the operators defined for set types in SOFL specifications and how test cases are generated from them.

(i) Membership (E_1 **inset** E_2)

This operator returns whether a given value,E_1, is contained in a given set, E_2, if E_1 is contained in E_2 then it returns **true**, else, **false**. For example, 7 **inset** $\{4, 5, 7, 9\}$ is **true**. The distance for it is defined as $card(E_2) - index(E_1, E_2)$[2]. In this case, the distance is 1, since the cardinality of $\{4, 5, 7, 9\}$ is 4 and 7 is the third element of $\{4, 5, 7, 9\}$.

Next, the test case generation method is shown. Consider a test condition, x **inset** S. First, when x is a variable, x is given a random value. Also, when S is a variable, give a random cardinality of S, but greater than or equal to 1. Next, we give the distance d. By solving $d = card(S) - index(x, S)$, we know the index of x in S. Finally, a test case can be generated by finding S such that the condition is satisfied. For example, consider 5 **inset** S. Since the length of S is undetermined, we give it a value of 4. Then, given a distance of 3, the index in S of 5 is 1, so the value of the first element of S is 5. The remaining three values of S are randomly given as $2, 4, and 6$, and there is a test case $S = \{5, 2, 4, 6\}$.

(ii) Non-membership (E_1 **notin** E_2)

This operator also returns whether a given value, E_1, is contained in a given set, E_2, but if E_1 is not contained in E_2 then it returns **true**, else, **false**. For example, 7 **notin** $\{4, 5, 9\}$ is **true**. The distance for it is defined as $card(E_2)$[2]. In this case, the distance is 3, since the cardinality of $\{4, 5, 9\}$ is 3.

Next, the test case generation method is shown. Consider a test condition, x **notin** S. First, when x is a variable, x is given a random value. Next, we give the distance d. According to the definition of distance, the cardinality of S is d. Finally, a test case can be generated by finding S such that conditions that cardinality is d and it does not contain x are satisfied.

For example, consider 5 **notin** S. Given a distance of 3, then the cardinality of S is 3. The remaining values of S are randomly given as $2, 4$ and 6, and there is a test case $S = \{2, 4, 6\}$.

(iii) Cardinality (**card**)

This operator returns the number of elements in a given set, as stated in the previous section 3.3.1.1. For example, $\mathbf{card}(\{5, 15, 25\})$ is 3. As it returns numeric types so we can treat the test condition containing this operator as a numeric expression so we can get the cardinality based on the traditional Vibration Method in section 2. It then generates sets that satisfy the obtained conditions about cardinality. The values of the elements of the set are given randomly.

(iv) Subset ($\mathbf{subset}(E_1, E_2)$)

This operator returns whether a given set, E_1 is subset of a given a set, E_2, if E_1 is subset of E_2 then it returns **true**, else, **false**. For instance, **subset** $(\{5, 15, 25\}, \{5, 10, 15, 20, 25\})$ is **true** and also **subset** $(\{5, 15, 25\}, \{5, 15, 25\})$ is **true**. The distance for it is defined as $card(E_2) - card(E_1)$. In these cases, the distance is 2 and 0, respectively.

Next, the test case generation method is shown. Consider a test condition, $\mathbf{subset}(S_1, S_2)$. At first, if S_1 and S_2 are variables, randomly give the length of S_2, n. Then, give the distance d, where d is less than or equal to $card(S_2)$ and is greater than or equal to zero. From the definition of distance, if S_1 and S_2 are variables, the cardinality of S_1 is determined as $n - d$, else, the cardinality of S_1 or S_2 is to be determined. Then, the set S_1 of the cardinality $n - d$ is randomly generated, and the set S_2 generates a set of cardinality n containing elements of S_1, but the remaining elements of S_2 can be random values.

For example, consider a test condition, $\mathbf{subset}(\{1, 2\}, S)$. Given a distance of 4, the cardinality of S is determined to be 6 from the definition of distance. Then, S_1 is the set that has the cardinality of 6 and also contains all elements in $\{1, 2\}$, that is, the test case S_1 is $\{1, 2, 5, 9, 3, 4\}$.

(v) Proper subset ($\mathbf{psubset}(E_1,\ E_2)$)

This operator returns whether a given set, E_1 is a proper subset of a given a set, E_2, if E_1 is a proper subset of E_2 then it returns **true**, else, **false**. For example, **subset** $(\{5, 15, 25\}, \{5, 10, 15, 20, 25\})$ is **true** and also **subset** $(\{5, 15, 25\}, \{5, 15, 25\})$ is **false**. The distance for it is defined as $card(E_2) - card(E_1)$. In these cases, the distance is 2 and 0, respectively.

The test case generation procedure is the same as the **subset** test case generation procedure. However, the distance d given must be positive, zero is not allowed.

(vi) Union ($\mathbf{union}(E_1, E_2)$)

This operator returns the union set of the two sets of arguments, E_1 and E_2. For example, $\mathbf{union}(\{1, 2, 3\}, \{2, 4, 6\})$ is $\{1, 2, 3, 4, 6\}$. The distance for it is the same as the distance in section 3.3.1.1 since the return value is a set type.

The following is an example of test case generation. For example, consider the test condition, $\mathbf{union}(S_1, S_2) = \{5, 10, 15, 20, 25\}$. The rerational operator is an equal sign so the distance for the cardinality is 0 and $\{0, 0, 0, 0, 0\}$. The value of $\mathbf{union}(S_1, S_2)$ is of course $\{5, 10, 15, 20, 25\}$. Then S_1 is a set, of elements randomly extracted from $\{5, 10, 15, 20, 25\}$, and S_2 is added to the elements not extracted in S_1, as well as elements randomly extracted from $\{5, 10, 15, 20, 25\}$. For instance, S_1 is $\{5, 25\}$ and S_2 is $\{10, 15, 20, 5\}$, and these become test cases.

(vii) Intersection ($\mathbf{inter}(E_1, E_2)$)

This operator returns the intersection of two sets, E_1, E_2, denoted by $E_1 \cap E_2$. For instance, $\textbf{inter}(\{1,2,3,4\},\{2,4,6,8\})$ is $\{2,4\}$. It returns also a set so the distance for it is the same as the distance in section 3.3.1.1.

The following is an example of test case generation. For example, consider the test condition,$\textbf{inter}(S_1, S_2) = \{1,2,3\}$. The rerational operator is an equal sign so the distance for the cardinality is 0 and $\{0,0,0\}$. The value of $\textbf{inter}(S_1, S_2)$ is of course $\{1,2,3\}$. Then, S_1 and S_2 have to have elements of $\{1,2,3\}$. And also, S_1 can have other elements, for example. 5 and 7. On the other hand, S_2 can also have other elements but they must be different from S_1's elements, for example, 4. Then, we can get a test case, $S_1 = \{1,2,3,5,7\}$ and $S_2 = \{1,2,3,4\}$.

(viii) Difference ($\textbf{diff}(E_1, E_2)$)

This operator returns the set of elements in E_1 that are not in E_2. For example, $\textbf{diff}(\{1,2,3,4\},\{2,4,6,8\})$ is $\{1,3\}$, and $\textbf{diff}(\{2,4,6,8\},\{1,2,3,4\})$ is $\{6,8\}$. The distance for it is the same as the distance in section 3.3.1.1 since the return value is a set type.

The following is an example of test case generation. For example, consider the test condition,$\textbf{diff}(S_1, S_2) = \{5,10\}$. The rerational operator is an equal sign so the distance for the cardinality is 0 and $\{0,0\}$, and $\textbf{diff}(S_1, S_2)$ is of course $\{5,10\}$. At first, S_1 must have elements of $\{5,10\}$. And then, S_1 and S_2 can add some elements other than $\{5,10\}$, for example, $1,2$ and 3. And then S_1 is the set of $\{5,10,1,2,3\}$ and S_2 is the set of $\{1,2,3\}$, and these become test cases.

3.3.2 Sequence types

A sequence is an ordered collection of objects that allows multiple occurrences of the same object. As with sets, the objects are known as elements of the sequence. For instance, "$[5,9,10]$."

In addition, in this study, the data type of an object or elements of a sequence is considered to be numeric types.

3.3.2.1 Distance Defined on Sequence types

In this sub-section, we explain how we define the distance for sequence types. We have defined the distance for sequence types as well as set types. The distance for sequence is $len(E_1) - len(E_2)$ and $[d_1, d_2, ..., d_n]$ for the test condition $E_1 <> E_2$. Where len means the number of the elements in the set or the length, and n is the length with the greater length in E_1 and E_2, and d_i is $e_{1_i} - e_{2_i}$, however when i is greater than $len(E_1)$ or $len(E_2)$ then d_i is just $-e_{2_i}$ or e_{i_j}, where e_{1_i} is the j-th element in E_i.

The following two examples are shown. First, for the test condition $[2,3] <> [0,2,4,6,8]$, distance is $[2,1,-4,-6,-8]$ and -3 with regard to the length. Then, for the test conditions $[0,2,4,6,8] <> [2,3]$, distance is $[-2,-1,0,4,6,8]$ and 3 with regard to the length.

The handling of the distance is the same as for distance for set types.

3.3.2.2 Test Case Generation for Sequence types

This sub-section describes how test cases are generated. The way test cases are generated in sequence types is almost the same as the way test cases are generated in set types, except that duplicate elements do not have to be removed from the generated sequence.

For example, consider the test condition, $S <> [1,3,5,7,9]$. And if the distance with respect to length is given as -3, the length of S is 2. The distance with respect to the element values is given as $[2,4,-5,-7,-9]$, where the last 3 elements correspond to the last 3 elements in $[1, 3, 5, 7, 9]$ multiplied by a minus. Then, S becomes the set of $[3, 7]$, and this is a test case.

This is the approach to test case generation for sequence types. Some operators are defined for sequence types in SOFL specifications. They are shown in the following sub-sections.

3.3.2.3 Operators in Sequence types

In this sub-section, we describe the operators defined for sequence types in SOFL specifications and how test cases are generated from them.

(i) Length ($\mathbf{len}(E)$)

This operator shows the number of its elements. For instance, $\mathbf{len}([5, 10, 15])$ is 3. As it returns numeric types so we can treat the test condition containing this operator as a numeric expression so we can get the length based on the traditional Vibration Method in section 2. It then generates a sequence that meets the obtained conditions about the length. The values of the elements of the sequence are given randomly.

(ii) Sequence application ($E(n)$)

This operator returns the element occurring at the position indicated by the given index n. For example, Let S be $[3, 6, 9, 12]$, then $S(2)$ is 6 as the 2nd element of S is 6. Since it also returns numeric types so we can treat the test condition containing this operator as a numeric expression so we can get the element of the given index based on the traditional Vibration Method in section 2. It then generates a sequence that satisfies the obtained conditions about the element. The length of the sequence and other values other than the index given are arbitrary.

(iii) Subsequence ($E(n, m)$)

This operator returns a sequence consisting of the nth to mth elements of the sequence, E. For example, Let S be $[5, 10, 14, 15]$, then $S(2, 4)$ is $[10, 14, 15]$. As it returns sequence types so we have to treat the test condition containing this operator as sequence expression. However, when deciding the distance, the distance regarding the length is 0, regardless of the relation operator.

We describe an example of generating test cases from this operator. Consider the test condition, $S(2, 4) <> [5, 7, 3]$. For instance, the distance is set as $[3, 4, 5]$. Then, the value of $S(2, 4)$ is $[8, 11, 8]$. The length of S is optional but must be greater than or equal to 4. Here it is set to 6, and the remaining elements can be any number, S being $[0, 8, 11, 8, 10, 20]$.

(iv) Head ($\mathbf{hd}(E)$)

This operator returns the first element of the sequence, E. For instance, Let S be $[4, 6, 1, 10, 2]$, then $\mathbf{hd}(S)$ is 4. Since it returns numeric types so we can treat the test condition containing this operator as a numeric expression so we can get the element of the given index based on the traditional Vibration Method in section 2. It then generates a sequence that satisfies the obtained conditions about the element. The length of the sequence and other values other than the index given are arbitrary.

(v) Tail ($\mathbf{tl}(E)$)

This operator returns a sequence with the head element of the set, E, removed. For example, Let S be $[3, 5, 5, 10, 4]$, then $\mathbf{tl}(S)$ is $[5, 5, 10, 4]$.

Since it returns sequence types so we have to treat the test condition containing this operator as sequence expression. We show the example of test case generation from this operator. Consider the test condition, $\mathbf{tl}(S) <> [1,2,3]$. Then, we give it the distance, 2, and $[2,4,3,5,0]$. $\mathbf{tl}(S)$ will be $[3,6,6,5,0]$ based on the distance. Finally, we can set the first element of S, for example, 5. Then, we can get the test case, $S = [5,3,6,6,5,0]$.

(vi) Concatenation ($\mathbf{conc}(E_1, E_2)$)

This operator returns a sequence that concatenates two sequences of arguments, E_1 and E_2. For example, $\mathbf{conc}([5,3,0],[2,2,4])$ is $[5,3,0,2,2,4]$.

Since it returns sequence types so we have to treat the test condition containing this operator as sequence expression. We show the example of test case generation from this operator. Consider the test condition, $\mathbf{conc}(S_1, S_2) <> [3,5,6,1,10]$. Then, we give it the distance, 2 and $[3,4,3,0,1,5,9]$. $\mathbf{conc}(S_1, S_2)$ will be $[6,9,9,1,11,5,9]$ based on the distance. This sequence needs to be separated, but at an arbitrary location, for example between the third and fourth element. Finally, we can get a test case, $S_1 = [6,9,9]$ and $S_2 = [1,11,5,9]$.

3.3.2.4 Test Case Generation from Conjunction about Sequence types

In this subsection, we introduce how test cases can be generated from conjunction that is not from a single atomic predicate but multiple atomic predicates.

Test conditions are often expressed in conjunction. This means that generating test cases from only one atomic predicate often results in test cases that do not satisfy the other conditional expressions that make up the test condition. We, therefore, offer an approach to generate test cases from conjunctions to solve this situation.

We deal with this problem by using a **Dummy Sequence**. The dummy sequence has undefined lengths and values of elements. An image of the dummy sequence is shown in Figure 3.2.

The test conditions are scanned one by one and values are determined for dummy sequences to satisfy the conditions. Test cases are generated based on the following three steps

First, test conditions with equal relational operators are processed.

Second, test conditions with the length statements are processed. At this time, the length of the dummy sequence is basically determined based on the test condition including the length operator, but also paying attention to the subscripts of subsequence operators and sequence application operators to determine the length to be longer than that. If there is no description of the length, the distance is determined randomly.

Finally, the test conditions with descriptions for each element of the sequence are examined to determine their value of them. Test conditions containing sequence application operators, head operators, tail operators, subsequence operators, etc. are processed to do that.

Then, if an element has an empty value, it is given a random value to make the dummy sequence a complete sequence. In this way, the length of the dummy sequence and the values of the elements are determined. This then becomes the test case.

Next, an example is given of generating test cases from conjunctions. The test condition is a conjunction consisting of atomic predicates shown in Listing 3.1.

At first, we have to prepare a dummy sequence.

Then, the atomic predicate that has an equal sign will be processed. There is an atomic predicate, $S(2,3) = [6,9,9]$, so the 2nd and 3rd elements of the dummy sequence are determined as 20 and 30 respectively.

Next, the atomic predicates including a description of the length are focused. According to the test condition, two atomic predicates are containing a length operator. Furthermore, looking at the arguments for sequence application and subsequence, there is a value of 5, so the length must be greater than or equal to 5. Arranging these, the length must be greater than 5 and less than 10. Based on the traditional vibration method, e.g. by giving a distance 2 to $5 < len(S) < 10$, the length is determined to be 7.

Then, the atomic predicates have descriptions of the value of elements. There are two atomic predicates about the element's value for the 1st element, $hd(S) > 10$ and $S(1) < 20$. The 1st element's value is 13 by giving the distance 3 to $10 < S(1) < 20$ based on the vibration method.

There is also an atomic predicate for the 5th element's value, $S(5) < 40$. By providing this atomic predicate with, for instance, a distance of 7, the value of the 5th element is 33.

Finally, the empty elements can get random values. There are still three empty elements,4th, 6th, and 7th, in the dummy sequence. These elements are randomly given as 7, 34, and 20 respectively.

Then the test case, $S = [13, 20, 30, 7, 33, 34, 20]$, is generated. These steps of test case generation from conjunction are also shown in Figure 3.3.

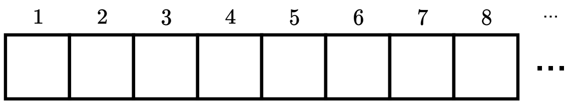


Figure 3.2: An image of a Dummy Sequence

Program 3.1: An example of Test Condition about Sequence Conjunction

```
len(S) < 10 \\means that the length of S is less than 10.
len(S) > 5 \\means that the length of S is greater than 5.
hd(S) > 10 \\means that the 1st element of S is greater than 10.
S(1) < 20 \\means that the 1st element of S is less than 20.
S(5) < 40 \\means that the 5th element of S is less than 40.
S(2,3) = [20,30] \\means that the subsequence of S from 2nd to 3rd is [20,30].
```

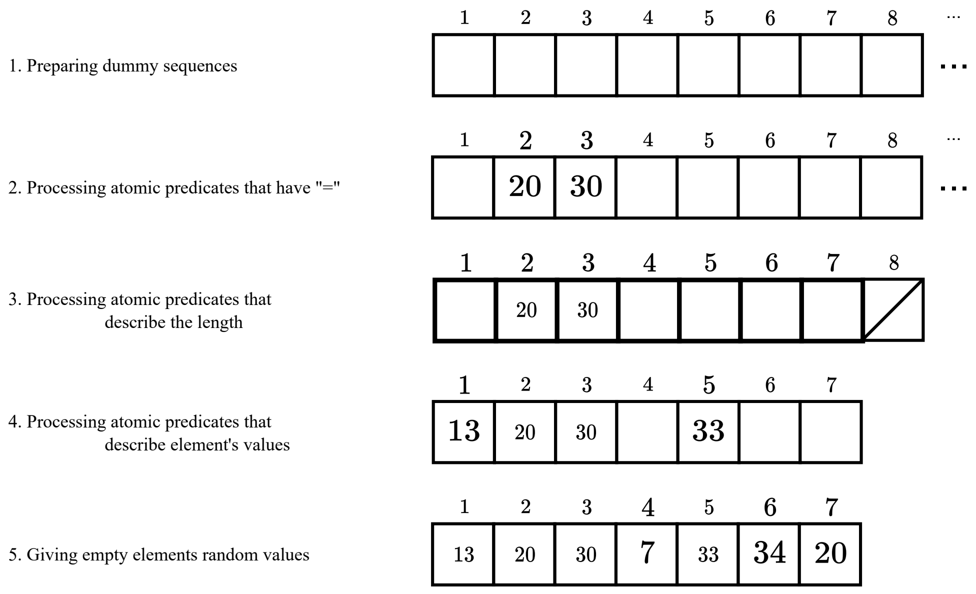

1. Preparing dummy sequences
1 2 3 4 5 6 7 8 ...
...
2. Processing atomic predicates that have "="
1 2 3 4 5 6 7 8 ...
20 30
...
3. Processing atomic predicates that describe the length
1 2 3 4 5 6 7 8
20 30
4. Processing atomic predicates that describe element's values
1 2 3 4 5 6 7
13 20 30 33
5. Giving empty elements random values
1 2 3 4 5 6 7
13 20 30 7 33 34 20

Figure 3.3: The flow of Test Case generation using Dummy Sequence

3.3.3 Composite types

A composite object usually contains several fields, each describing the different aspects of the object. A composite object is like a record in Pascal and a structure in C.

The composite type A is declared in the form shown in Listing 3.2. Where f_i $(i = 1, ..., n)$ are variables called fields and T_i are their types. Each field represents an attribute of the composite object of the type.

For example, Car of the composite type that has fields of weight, fuel, speed, and location is declared in the form shown in Listing 3.3. Furthermore, only one constructor known as make-function is available for composite types. For example, it is **mk_Car(1000, 30, 50, 0)**. The make-function yields a composite value of a composite type Car whose field values are $1000, 30, 50$, and 0 which corresponds to field weight, fuel, speed, and location, respectively.

And, there is an operator in composite types. It is the **field select** operator. Let a be a variable of composite type A. Then, we use $a.f\_i$ to represent the field, f_i $(i = 1, ..., n)$, of the composite object a.

In addition, in this study, the data type of fields of composite types is considered to be numeric types.

Program 3.2: The general form of a composite type declaration

```
A = composed of
    f_1: T_1
    f_2: T_2
     ...
    f_n: T_n
end
```

Program 3.3: Declaration of composite type Car

```
Car = composed of
    weight: real
    fuel: real
    speed: rael
    location: real
end
```

3.3.3.1 Distance Defined on Composite types

In this subsection, we show how we define the distance for composite types. We have also defined the distance for composite types as well as set types and sequence types. The distance for composite types focuses on the difference in the values of each field. It is defined as $(v_{1_1} - v_{2_1}, v_{1_2} - v_{2_2}, ..., v_{1_n} - v_{2_n})$ where v_{1_i} is the value of the i-th field of the composite type on the left-hand side and on the other hand, v_{2_i} is the value of the i-th field of the composite type on the right-hand side.

One example is given next. For example, for the test condition, mk_Car$(1000, 30, 50, 0) <>$ mk_Car$(1000, 25, 80, 100)$, the distance is $(0, 5, -30, -100)$.

The handling of the distance is almost the same as for distance for set types and sequence types, however, there is no distance for the length or the cardinality, since the number of the fields of composite types must correspond.

3.3.3.2 Test Case Generation for Composite types

In this subsection, we introduce the methods to generate test cases from composite types. It is the almost same way in set types as well as composite types.

We present three examples below. These are about the composite type Car in Listing 3.3.

For example, consider the test condition, $car <>$mk_Car$(1000, 30, 50, 0)$. And if the distance,$(100, -20, -5, 10)$, is given, then, car is mk_Car$(1100, 10, 45, 10)$ based on the definition of the distance for composite types.
And for instance, consider the test condition, $car1 <> car2$. At first, $car1$ is randomly given mk_Car$(1500, 10, 100, 100)$. Then, the distance is given, for example $(-300, 5, -20, 0)$. A $car2$ will be mk_Car$(1200, 15, 80, 100)$. Finally, we can get test cases, $car1 =$ mk_Car$(1500, 10, 100, 100)$ and $car2 =$ mk_Car$(1200, 15, 80, 100)$.
Thirdly, for example, we consider the test condition, $car.speed > 40$. Based on the previous vibration method, we give it the distance, 25, then, car's speed is 65. Then, car is mk_Car$(500, 40, 65, 90)$, where values for fields other than speed are randomly generated.

3.3.3.3 An Operator in Composite types

In this sub-section, we describe the operator defined for composite types in SOFL specifications and how test cases are generated from it

The operator is field modification operator (**modify**). Given a composite value, say co, of type A, we can apply the field modification operator modify to create another composite value of the same type. For example, let $car1 =$ mk_Car$(1500, 10, 100, 100)$, then, we can have the expression: $car2 = \textbf{modify}(car, speed- > 0, location- > 10)$. Then, $car2$ shows mk_Car$(1500, 10, \mathbf{0}, \mathbf{10})$. This operator does not show the conditional expression, so we do not have to generate test cases from it. It has to be just modified the fields.

3.3.3.4 Test Case Generation from Conjunction about Composite types

This sub-section describes how test cases can be generated from conjunction consisting of multiple atomic predicates.

This is similar to when generating a test case from a conjunction of sequence types, for a composite type that does not have field values, the values of each field are determined so that the test conditions are met. In the case of sequence types, the length has to be determined, but in this case, this is not necessary as the number of fields is predefined. The test case generation procedure is completed in three steps.
First, an atomic predicate containing an equal sign is processed.
Then, afterward, the test conditions for field values are processed, and the field values are determined based on the vibration method.
And finally, if there are any undetermined fields, they are given a random value.

Next, an example is given of test case generation from a conjunction. The test condition is the conjunction consisting of atomic predicates shown in Listing 3.4. The composite type is Car type in Listing 3.3.
First of all, prepare car of Car types that do not have any values of fields.
Then, the atomic predicate that has an equal sign can be processed. There is an atomic predicate, $car.weight = 1000$, so $car'sweight$ get the value of 1000.
Next, the atomic predicates including a description of the field's values will be processed. These four atomic predicates are about the field's values in the test condition. These describe fields of $fuel$ and $speed$. For the field of $fuel$, there are two atomic predicates, $car.fuel > 0$ and $car.fuel <= 30$. Using the vibration method, for instance, by providing a distance 20 to $0 < car.fuel <= 30$, $car'sfuel$ is determined to be 20.
Also, for the field of $speed$, there are two atomic predicates, $car.speed > 80$ and $car.speed <= 110$. Based on the vibration method, for example, by giving a distance 35 to $80 < car.fuel < 110$, $car'sspeed$ is 115.
Finally, the empty fields can get random values. There is an empty field, $location$, in car. It is randomly given

as 25. Then the test case, $car = \text{mk\_}car(1000, 20, 115, 25)$, is generated. These steps of test case generation from conjunction are also shown in Figure 3.4.

Program 3.4: An example of Test Condition about Composite type *Car* Conjunction

```
car.weight = 1000 \\means that car's␣weight␣is␣1000kg.
car.speed␣>␣80␣\\means␣that␣car's speed is greater than 80km/h.
car.speed<110 \\means that car's␣speed␣is␣less␣than␣110km/h.
car.fuel␣>␣0␣\\means␣that␣car's fuel is greater than 0L.
car.fuel <= 30 \\means that car's␣fuel␣is␣less␣than␣30L.
```

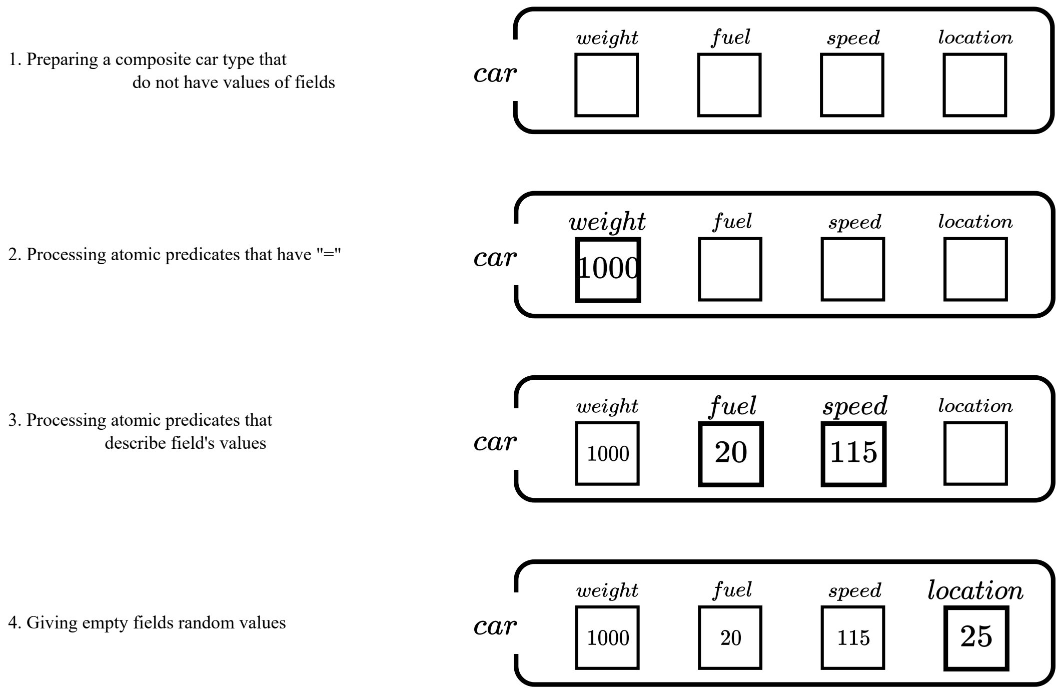

1. Preparing a composite car type that do not have values of fields
car
weight
fuel
speed
location
2. Processing atomic predicates that have "="
1000
3. Processing atomic predicates that describe field's values
20
115
4. Giving empty fields random values
25

Figure 3.4: The flow of Test Case generation from Composite *Car* type Conjunction

Chapter 4

A Supporting Tool

In this section, we introduce our supporting tool for the Vibration method. This tool provides services for generating test cases, automatic theorem proving, and test results analysis. The tool is composed of several layers and each of them is described below.

4.1 Core Layer

This layer is responsible for carrying out syntactical analysis of SOFL specifications and converting the specification into a Functional Scenarios Form (FSF). An FSF of a specification is a disjunction of functional scenarios. Each functional scenario is a conjunction of two conditions, known as the *test condition* and the *defining condition*. The test condition is a conjunction of atomic predicates that contain only input variables and serve as the basis for generating test cases, while the defining condition must contain the output variables and plays the role of defining the output variables using the input variables. The tool can obtain all of the atomic predicates of any given test condition for test case generation.

Furthermore, the tool now supports set types, sequence types, and composite types, so that the operators defined for them can be recognized. And in order to handle composite types, in particular, it is possible to read the names and field information of composite data types from the specification. When dealing with composite types in the test conditions of a specification, it is necessary to describe the declaration of the composite type to be used in addition to the test conditions and definition conditions in the specification. The data type declarations are shown in Section 3.3.3.

4.2 Service Layer

The service layer of the tool provides the following four major functions: defining distance, automatic test case generation, automatic theorem proving and test result evaluation The test cases about set types, sequence types and, composite types can be generated.

4.2.1 Defining distance

After obtaining an atomic predicate, the tool can support the setting of a distance for vibration in test case generation. The distance can be set manually by the user through the GUI, but it can also be automatically set by the tool system. However, as it has not yet been fully discussed how distances should be given in set types, sequence types and composite types, this tool gives distances randomly.

4.2.2 Automatic Test Case Generation

This function is one of the most important functions of the tool. Selecting an atomic predicate from a designated test condition written in the SOFL specification language, the tool can automatically generate test cases from the atomic predicate. Currently, in addition to numerical types, the system can handle atomic predicates, including set types, sequence types and composite types expressions. The specific method for test case generation is the bisection method, as introduced before. Furthermore, test cases can be generated from the conjunction of several of those atomic predicates, excluding set types. However, there are certain restrictions here. That is, variables, elements of sets and sequences, and fields of composite types must be independent. For example, it is still not possible to generate a test case from conjunction such as $S(1) + S(2) = 10 \ \wedge \ S(1) < 10$ or $car.speed = car.location/time \ \wedge \ time > 0$.

4.2.3 Automatic Theorem Validation for numeric types

This is one of the most important functions of the tool too. Given a theorem composed of a hypothesis and a conclusion, the tool can automatically verify the theorem based on testing. That is, a test case, containing values for all the variables involved in both the hypothesis and the conclusion, is first automatically generated to satisfy the hypothesis, and then the test case is used to evaluate the conclusion. If the hypothesis evaluates to true but the conclusion evaluates to false, then, it implies that a bug exists in the theorem. Repeating this process to generate as many test cases as necessary and to evaluate the hypothesis and the conclusion, we will be able to find out whether the theorem holds or not. This function support only numeric types, and other data types are not supported.
The theorem shown in Listing 4.1 is an example written in SOFL. The hypothetical condition part is the part of the statement before "|−", and the conclusion part is the part of the statement after "|−". Test cases can be generated from the hypothetical condition part. Test cases generated from this part and the results are shown in Table 4.1. The number of vibrations is set to 6 and the amplitude of the distance vibration is set to 0.5. We can prove that the theorem is false since there is a FALSE result on the second row of Table 4.1.

Program 4.1: An example of a theorem written in SOFL

```
a >= 0 and b >= 0 |- a*a + b*b >= a + b
```

4.3 Management Layer

The tool can read SOFL specifications from the local computer. The generated test cases can be saved in CSV file format. The saved test cases can also be read and used for verifying test results. The tool can display test conditions as a list of atomic predicates it involves, the determined distances, and the generated test cases in a comprehensible manner through the GUI of the tool.

4.3.1 Automatic Test Results Evaluation

The concept of automatic theorem validation is also used to analyze test results. As introduced previously, in the Vibration method for testing, we generate test cases from the test condition. To determine whether the test case finds a bug in the program, we need to use the mechanism similar to a theorem for a decision. In other words, we treat the test condition of the functional scenario as the hypothesis of a theorem and the defining condition of the functional scenario as the conclusion of the theorem. In the defining condition, not only the

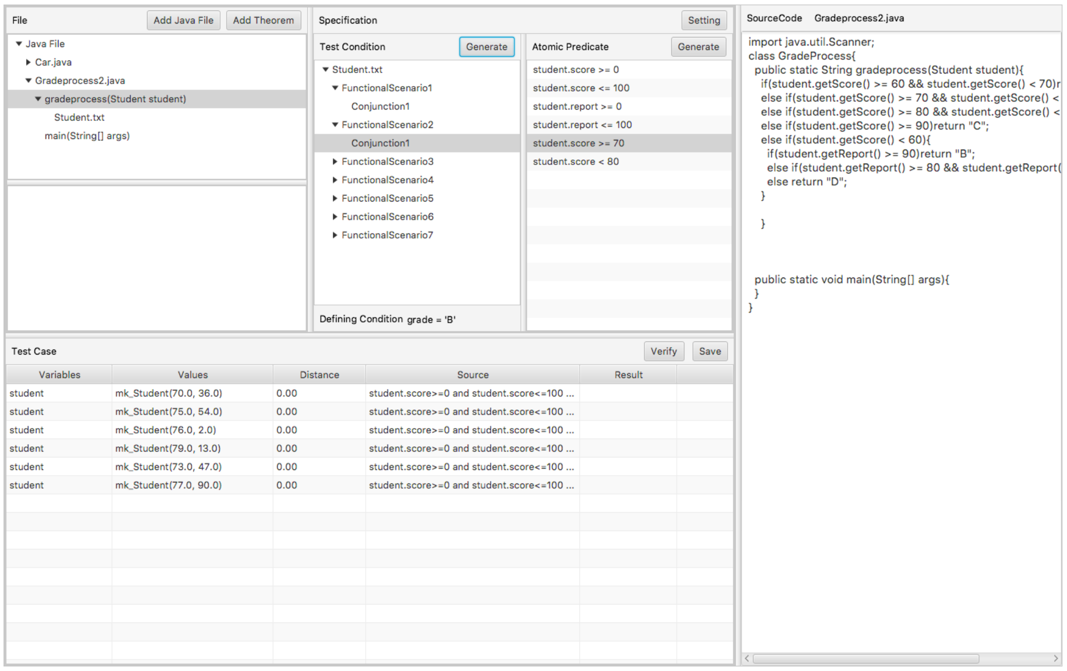
Table 4.1: A generated test set and the result

	a	b	RESULTS
1	0	0	TRUE
2	0.5	0.5	FALSE
3	1	1	TRUE
4	1.5	1.5	TRUE
5	2	2	TRUE
6	2.5	2.5	TRUE

input variables are involved, but the output variables are also involved. After running the program with the test case generated from the test condition, we will be able to obtain the values for all the output variables. Thus, we will be able to substitute the values for the output variables occurring in the defining condition to evaluate it to be true or false. If it is false, it proves that a bug of the program is found. Also, this function supports only numerical types as well as the theorem validation.

4.4 GUI Layer

The overall GUI of the tool is shown in Figure 4.1. The GUI of the tool consists of four mainframes. First of all, in the upper left frame, you can input files. By pressing the appropriate button at the top of this frame, SOFL specifications and theorems are loaded. Next, the right frame shows functional scenarios contained in the loaded specifications. Depending on the selected conjunction, the subframe to the right of it shows the atomic predicates that make up the conjunction. In the lower part of the frame, the defining condition of the functional scenario or the conclusion of the theorem is displayed. Furthermore, in this frame, test cases are generated based on the selected conjunction or atomic predicate. The generated test cases are kept in a table in the frame below it. If you want to save the test cases in CSV format on your local computer, you can do by pressing the button at the top right of this frame. If you want to verify the test results or theorems, you can do it by clicking on the button next to the save button.



```
▼ Java File
  ▶ Car.java
  ▼ Gradeprocess2.java
    ▼ gradeprocess(Student student)
        Student.txt
      main(String[] args)
```

```
▼ Student.txt
  ▼ FunctionalScenario1
      Conjunction1
  ▼ FunctionalScenario2
      Conjunction1
  ▶ FunctionalScenario3
  ▶ FunctionalScenario4
  ▶ FunctionalScenario5
  ▶ FunctionalScenario6
  ▶ FunctionalScenario7
```

Defining Condition grade = 'B'

```
student.score >= 0
student.score <= 100
student.report >= 0
student.report <= 100
student.score >= 70
student.score < 80
```

Variables	Values	Distance	Source	Result
student	mk_Student(70.0, 36.0)	0.00	student.score>=0 and student.score<=100 ...	
student	mk_Student(75.0, 54.0)	0.00	student.score>=0 and student.score<=100 ...	
student	mk_Student(76.0, 2.0)	0.00	student.score>=0 and student.score<=100 ...	
student	mk_Student(79.0, 13.0)	0.00	student.score>=0 and student.score<=100 ...	
student	mk_Student(73.0, 47.0)	0.00	student.score>=0 and student.score<=100 ...	
student	mk_Student(77.0, 90.0)	0.00	student.score>=0 and student.score<=100 ...	

```
import java.util.Scanner;
class GradeProcess{
  public static String gradeprocess(Student student){
    if(student.getScore() >= 60 && student.getScore() < 70)r
    else if(student.getScore() >= 70 && student.getScore() <
    else if(student.getScore() >= 80 && student.getScore() <
    else if(student.getScore() >= 90)return "C";
    else if(student.getScore() < 60){
      if(student.getReport() >= 90)return "B";
      else if(student.getReport() >= 80 && student.getReport(
      else return "D";
    }

    }



  public static void main(String[] args){
  }
}
```

Figure 4.1: The snapshot of the supporting tool

Chapter 5

A Small Experiment

This section presents small experiments we have conducted to evaluate the major functions of our tool. We have conducted two types of experiments. The first is to check the effectiveness of the vibration method by performing mutation tests on numerical types. The other is to verify whether the correct test case generation can be performed for non-numeric types. Those are described in more detail in the following sub-sections.

5.1 Experiment I

This experiment is conducted to evaluate the major functions of our tool, generating test cases from especially numeric types.
We use mutation testing [4] as the means to verify whether the test cases generated using our tool are effective in detecting bugs in programs. A program for grading students' examinations is used to create mutants. Each mutant contains one inserted bug. There are three rules for the generation of mutants and the rules for inserting a mutant are shown in Table 5.1. In Table 5.1, LCR is short for Logical Connector Replacement, ROR for Relational Operator Replacement and, UOI for Unary Operator Insertion. And also, Table 5.1 is just an example, and these operators apply equally well to OR operations as it does to AND operations.

The program for grading determines the grade as the output according to the score and the test of the inputs. A SOFL specification of this program is shown in Listing 5.1. Two programs are written to implement this specification and they are used to create mutants, respectively. One program, called Program A, is written in a style that is similar to that of the specification. The other program, called Program B, is implemented in a way that refines the logical expressions of the specification into more detailed program paths. The purpose of using these two different style programs is to help observe the effect of our Vibration method in different implementation situations.

In order to compare the effectiveness of test cases generated by the tool, we also prepare two other types of test sets. One is based on Boundary Value Analysis and the other is based on random testing. Boundary Value Analysis is one of the commonly used black-box testing techniques. Random testing uses randomly produced test cases to test the program and its advantage over many other existing methods is the low cost.

We conduct this experiment by generating two test sets for each of the three testing techniques, vibration method, boundary value analysis, and random testing. A summary of the six test sets is given in Table 5.2. The first is generated based on the vibration method. This test set consists of 40 test cases. The specification, as shown in Listing. 5.1, consists of 10 conjunctions, and the test cases were generated by vibrating four times for each conjunction. The second is based on the vibration method too. This test set consists of 90 test cases.

These were generated by vibrating nine times for each conjunction. The third is based on boundary value analysis. This test set has 40 test cases. As mentioned before, the specification consists of 10 conjunctions, and it was generated by taking the maximum and minimum values for each range and considering the combination of the two variables, score and report. The fourth is also based on boundary value analysis. This test set is generated by taking the median value as a representative value in addition to the maximum and minimum values for each range. The total number of test cases is 90. The fifth and final are based on random testing. These were generated by generating random values for real values between 0 and 100, the pre-conditions of the specification. They consisted of 40 and 90 test cases respectively.

First of all, mutation testing is implemented on program A. The number of mutants created for this program is 359. The result is shown in Table 5.3. Comparing the vibration method with the boundary value analysis, it can be seen that they have equivalent accuracy. Here, accuracy means the ability or quality of the test cases used to detect bugs lurking in the program. This is because the minimum and maximum values in the boundary value analysis correspond to test cases of the vibration method which are generated from taking the minimum and maximum distances. However, the number of mutants killed by the test cases of the vibration method is one less than that of the mutants killed by the test cases of boundary value analysis. The reason is that the program has two input variables, and the test cases of boundary value analysis are generated by combining the maximum and minimum values for each variable, but in the vibration method, the test cases are generated without considering the combination of the maximum and minimum distances. In the case of random testing, it shows an accuracy of about 60% for all mutations when 90 cases are generated, but the accuracy of the vibration method is superior to the random testing.

Next, mutation testing is implemented on Program B. The number of mutants created for this program is 1099, as shown in Table 5.4. The test cases generated by the vibration method show the highest accuracy. Comparing the results of the vibration method with 40 test cases and the results of the boundary value analysis with 40 test cases, the vibration method shows 12% better accuracy than the boundary value analysis. Furthermore, comparing the results of the vibration method with 40 test cases and the results of the boundary value analysis with 90 test cases, the vibration method shows a slightly higher ratio of kills, indicating a higher accuracy with fewer test cases. This result shows that the vibration method is more effective when the program is a more detailed refinement of the specification. However, compared to the result of the mutation testing in the first program, the accuracy in this program is lower. This means that the accuracy of the vibration method depends on the program used and a new problem has arisen. On the other hand, in the random testing, the accuracy does not change significantly depending on programs, and shows a constant accuracy independent of the target program.

Table 5.1: Mutation Operators

NAME	EXAMPLE APPLYING OPERATORS
LCR	$a\&\&b \rightarrow \{a, b, a \mid\mid b, true, false\}$
ROR	$a > b \rightarrow \{a < b, a <= b, a >= b, true, false\}$
UOI	$a \rightarrow \{a++, a--\}$

Program 5.1: The SOFL specification of the Grades Processing System

```
GradeProcess (test,report:real) grade:char
pre test>=0 and test<=100 and report>=0 and report<=100
post test>=90 and report>=80 and grade='S'
     test>=80 and report>=70 and report<80 or test>=80 and test<90 and report>=80 and grade=
       'A'
     test>=70 and report>=60 and report<70 or test>=70 and test<80 and report>=70 and grade=
       'B'
     test>=60 and report>=50 and report<60 or test>=50 and test<70 and report>=60 and grade=
       'C'
     test<50 or report<50 or test>=50 and test<60 and report>=50 and report<60 and grade='D'
end_process
```

Table 5.2: A summary of the test sets

	TESTING METHODS	NUMBER of TEST CASES	NOTES
1	Vibration Method	40	Number of Vibrations: 4
2		90	Number of Vibrations: 9
3	Boundary Value Analysis	40	Values taken the max and min values for each range
4		90	Values taken the max, min and med values for each range
5	Random Testing	40	random values for real values between 0 and 100
6		90	

Table 5.3: Results of the mutation testing for target program A

TESTING METHODS	NUMBER of TEST CASES	NUMBER of KILLING (RATIO)
Vibration Method	40	276 (76.880%)
	90	276 (76.880%)
Boundary Value Analysis	40	277 (77.159%)
	90	277 (77.159%)
Random Testing	40	178 (49.582%)
	90	214 (59.610%)

The total number of mutants is 359.

Table 5.4: Results of the mutation testing for target program B

TESTING METHODS	NUMBER of TEST CASES	NUMBER of KILLING (RATIO)
Vibration Method	40	740 (67.334%)
	90	778 (70.792%)
Boundary Value Analysis	40	606 (55.141%)
	90	738 (67.152%)
Random Testing	40	515 (46.861%)
	90	623 (56.688%)

The total number of mutants is 1099.

5.2 Experiment II

This experiment demonstrates whether the tool can generate correct test cases based on the test conditions and the distance of the vibration method.
A total of 37 test conditions have been prepared. The list includes two for the set types. Two for each operator of set types. Two for sequence types. Two for each operator of sequence types. Five for the composite types. Furthermore, two for each conjunction of sequence and composite types.
The test conditions for set types, sequence types and composite types are given in Table 5.5, Table 5.6 and Table 5.7.

We generate five test cases from each test condition using our supporting tool. The test cases and the distances are shown in Table 5.8, Table 5.9 and Table 5.10.
Checking all these test cases confirmed that they all generated the correct test cases with no conflict in the definition of the distance.

Table 5.5: The Test Conditions for Set Types

No.	TEST CONDITION
1	$S1 <> S2$
2	$S1 <> \{a, b, c, 1, 2\}$
3	$7\ inset\ x = true$
4	$a\ inset\ \{2, 4, 8\} = true$
5	$a\ notin\ \{4, 5, 6, 7\} = true$
6	$4\ notin\ x = true$
7	$card(S1) = 3$
8	$card(S1) < 5$
9	$subset(\{1, 2\}, S1) = true$
10	$subset(\{5, 10\}, \{5, a, 25, 100\}) = true$
11	$psubset(\{1, 2\}, S1) = true$
12	$psubset(\{a\}, S1) = true$
13	$union(S1, S2) = \{1, 2, 3, 4, 5\}$
14	$union(\{5, a, b\}, \{a, c\}) = \{5, 10, 4, 3\}$
15	$inter(S1, S2) = \{\}$
16	$inter(S1, \{1, 3\}) = \{1\}$
17	$diff(S1, S2) = \{\}$
18	$inter(S1, \{a, b, 2, 4, 6\}) = \{3\}$

Table 5.6: The Test Conditions for Sequence types

No.	TEST CONDITION
1	$S1 <> S2$
2	$S1 <> [1, 2, 3, 4]$
3	$len(S1) >= 5$
4	$len(S1) = 3$
5	$S1(3) + S1(6) = 10$
6	$S1(3) + S1(6) - S1(1) + a - b > 10$
7	$S1(2, 3) <> [5, 10]$
8	$S1(1, 5) = [2, 4, 4, 5, 5]$
9	$hd(S1) > 5$
10	$hd(S1) = a$
11	$tl(S1) = []$
12	$tl(S1) <> [1, 2, 3]$
13	$conc(S1, S2) <> [2, 4, 6, 8, 10]$
14	$conc(S1, S2) = [1, 2, 3, 4, 5]$
15	$len(S1) = 7 \ \wedge \ S(1) >= 0 \ \wedge \ hd(S1) <= 10 \ \wedge \ S1(3, 6) = [3, 4, 5, 6]$
16	$len(S1) < 8 \ \wedge \ S1(1) > 0 \ \wedge \ hd(S1) < 10 \ \wedge \ S1(4) > 400 \ \wedge \ S1(4) < 500 \ \wedge \ S1(6) > 10$

Table 5.7: The Test Conditions for Composite types

No.	TEST CONDITION
1	$car1 <> \text{mk\_}Car(3000, 60, 120, 15)$
2	$car1 = \text{mk\_}Car(3000, a, b, 15)$
3	$car1.fuel > 10$
4	$car1.fuel > a$
5	$car1.speed > car1.location/time$
6	$car1 <> \text{mk\_}Car(0, 0, 0, 0) \ \wedge \ car1.fuel >= 0 \ \wedge \ car1.fuel < 30 \ \wedge \ car1.speed = 100 \ \wedge \ car1.location > 0$
7	$car1.fuel >= 0 \ \wedge \ car1.fuel < 50 \ \wedge \ car1.speed < 100 \ \wedge \ car1.speed > 0 \ \wedge \ car1.location > 0$ $\wedge \ car2 = modify(car1, fuel- > 0)$

Table 5.8: The Test Cases from Test Conditions in Figure5.5

No.	TEST CASE	DISTANCE
1	$S1 = \{10, 9, 1, -10, -7, 12, 14, 5, -13, 4\}, S2 = \{0, 17, 7\}$	$7, \{10, -8, -6, -10, -7, 12, 14, 5, -13, 4\}$
	$S1 = \{2, 12, 1, 27\}, S2 = \{9, 8, 10, 12, 3\}$	$-1, \{-7, 4, -9, 15, -3\}$
2	$S1 = \{-4, 7, -5, 8, -15\}, a = 4, b = 13, c = 5$	$2, \{-8, -6, -10, 7, -7, 7, -15\}$
	$S1 = \{11, 24, 13, -1, 16, 7, 15, 3, 9, -3, 5\}, a = 7, b = 14, c = 8$	$7, \{4, 10, 5, -2, 14, 7, 15, 3, 9, -3, 13, 5\}$
3	$S1 = \{7, 5, 2, 8, 0\}$	4
	$S1 = \{7\}$	0
4	$a = 2$	2
	$a = 4$	1
5	$a = 30$	4
	$a = 47$	4
6	$S1 = \{15, 2, 45, 25, 13\}$	5
	$S1 = \{23, 40, 32, 17, 39, 6, 2, 41, 5, 36\}$	10
7	$S1 = \{38, 49, 89\}$	0
	$S1 = \{38, 55, 23\}$	0
8	$S1 = \{83\}$	4
	$S1 = \{\}$	5
9	$S1 = \{1, 2, 5, 17, 14, 3\}$	4
	$S1 = \{1, 2, 12, 13, 6\}$	3
10	$a = 10$	2
	$a = 10$	2
11	$S1 = \{1, 2, 10, 19, 12, 18\}$	4
	$S1 = \{1, 2, 7, 12\}$	2
12	$a = 17, S1 = \{17, 7\}$	1
	$a = 6, S1 = \{6, 19, 14\}$	2
13	$S1 = \{1, 3, 4, 5, 2\}, S2 = \{2\}$	$0, \{0, 0, 0, 0, 0\}$
	$S1 = \{2, 5, 4\}, S2 = \{1, 3, 4\}$	$0, \{0, 0, 0, 0, 0\}$
14	$a = 4, b = 3, c = 10$	$0, \{0, 0, 0, 0\}$
	$a = 3, b = 10, c = 4$	$0, \{0, 0, 0, 0\}$
15	$S1 = \{11, 14, 19\}, S2 = \{18\}$	$0, \{\}$
	$S1 = \{14\}, S2 = \{\}$	$0, \{\}$
16	$S1 = \{1\}$	$0, \{0\}$
	$S1 = \{1, 9, 18\}$	$0, \{0\}$
17	$S1 = \{14\}, S2 = \{14, 19, 5, 3\}$	$0, \{\}$
	$S1 = \{2, 13\}, S2 = \{2, 13\}$	$0, \{\}$
19	$S1 = \{3, 17\}, a = 3, b = 8$	$0, \{0\}$
	$S1 = \{3\}, a = 9, b = 3$	$0, \{0\}$

Table 5.9: The Test Cases from Test Conditions in Figure5.6

No.	TEST CASE	DISTANCE
1	$S1 = [-3, 12, -9, -11, 6, -11, 8, 14, 7], S2 = [5]$	$8, [-8, 12, -9, -11, 6, -11, 8, 14, 7]$
	$S1 = [0, 5, 12, -4, 14, -8, 1], S2 = []$	$7, [0, 5, 12, -4, 14, -8, 1]$
2	$S1 = [1, -7, 6, -9, 5, 8, -5]$	$3, [0, -9, 3, -13, 5, 8, -5]$
	$S1 = [13, -11, -4, -6, 10]$	$1, [12, -13, -7, -10, 10]$
3	$S1 = [3, 12, 2, 5, 29, 20, 12]$	2
	$S1 = [12, 0, 5, 11, 8]$	0
4	$S1 = [8, 6, 3]$	0
	$S1 = [4, 10, 13]$	0
5	$S1 = [7, 0, -20, 12, 14, 30, 4]$	0
	$S1 = [0, 14, -13, 11, 12, 23]$	0
6	$a = 4, b = 22, S1 = [7, 13, 56.82, 3, 9, 0]$	21.82
	$a = 29, b = 5, S1 = [6, 8, 32.12, 0, 4, 1, 3, 5]$	41.12
7	$S1 = [0, 13, 5]$	$0, [8, -5]$
	$S1 = [9, -1, 13, 1]$	$0, [-6, 3]$
8	$S1 = [2, 4, 4, 5, 5, 3, 5, 8]$	$0, [0, 0, 0, 0, 0]$
	$S1 = [2, 4, 4, 5, 5, 3]$	$0, [0, 0, 0, 0, 0]$
9	$S1 = [12.91, 13, 0, 8, 13]$	7.91
	$S1 = [10.22, 18, 18, 9, 15, 14, 15]$	5.22
10	$a = 19, S1 = [19, 5, 4, 8]$	0
	$a = 21, S1 = [21, 7, 1, 19, 8, 17, 10, 8, 9, 17, 9, 1, 10, 16]$	0
11	$S1 = [3]$	$0, []$
	$S1 = [2]$	$0, []$
12	$S1 = [5, -5, 11, 11, 1, 11, -8, -4]$	$4, [-6, 9, 8, 1, 11, -8, -4]$
	$S1 = [7, 1, 4, 10, -3, -4, -14, -11, -1]$	$5, [0, 2, 7, -3, -4, -14, -11, -1]$
13	$S1 = [11, -7, -7, -5, 10], S2 = [-8, 8, -15, -7, 6, 13]$	$6, [9, -11, -13, -13, 0, -8, 8, -15, -7, 6, 13]$
	$S1 = [1, 9, -5, 4, 9, 0, 10, -5], S2 = [-2, 2, 14]$	$6, [-1, 5, -11, -4, -1, 0, 10, -5, -2, 2, 14]$
14	$S1 = [1, 2, 3, 4], S2 = [5]$	$0, [0, 0, 0, 0, 0]$
	$S1 = [1], S2 = [2, 3, 4, 5]$	$0, [0, 0, 0, 0, 0]$
15	$[2.164, 2.606, 3.000, 4.000, 5.000, 6.000, 0.154]$	$0, [2.164, none, 0, 0, 0, 0, none]$
	$[3.385, 11.957, 3.000, 4.000, 5.000, 6.000, 1.362]$	$0, [3.385, none, 0, 0, 0, 0, none]$
16	$[0.549, 0.520, 4.222, 426.947, 0.000, 43.400]$	$2, [0.549, none, none, 26.947, none, 33.4]$
	$[5.951, 5.462, 5.502, 453.467, 5.688, 27.600, 2.163]$	$1, [5.951, none, none, 53.467, none, 17.6, none]$

Table 5.10: The Test Cases from Test Conditions in Figure5.7

No.	TEST CASE	DISTANCE
1	$car1 = \text{mk\_}Car(3033.56, 96.24, 130.19, 30.08)$	$(33.56, 36.24, 10.19, 15.08)$
	$car1 = \text{mk\_}Car(3011.97, 101.89, 166.03, 40.64)$	$(11.97, 41.89, 46.03, 25.64)$
2	$car1 =\text{mk\_}Car(3000.000, 3.000, 0.000, 15.000), a = 3.000, b = 0.000$	$(0, 0, 0, 0)$
	$car1 = \text{mk\_}Car(3000.000, 11.000, 22.000, 15.000), a = 11.000, b = 22.000$	$(0, 0, 0, 0)$
3	$car1 = \text{mk\_}Car(29, 32.1, 7, 0)$	22.10
	$car1 =\text{mk\_}Car(26, 54.04, 8, 24)$	44.04
4	$car1 = \text{mk\_}Car(27.000, 58.900, 24.000, 0.000), a = 23.000$	35.90
	$car1 = \text{mk\_}Car(19.000, 22.190, 13.000, 2.000), a = 18.000$	4.19
5	$car1 = \text{mk\_}Car(4, 17, 1.513, 188), time = 17$	7.72
	$car1 = \text{mk\_}Car(27, 8, 4.82, 26), time = 15$	46.30
6	$car1 = \text{mk\_}Car(8, 20., 100, 14.97)$	$(none, 20, 0, 14.97)$
	$car1 = \text{mk\_}Car(5, 16.0, 100, 26.92)$	$(none, 16, 0, 26.92)$
7	$car1 = \text{mk\_}Car(15, 40, 11, 14.47), car2 = \text{mk\_}Car(15, 0, 11, 14.47)$	$(none, 40, 11, 14.47)$
	$car1 = \text{mk\_}Car(0, 23, 33, 37.22), car2 = \text{mk\_}Car(0, 0, 33, 37.22)$	$(none, 23, 33, 37.72)$

Chapter 6

Related Work

There are several related works of generating test cases from formal specifications. They include test case generation by the combination of random testing and decision table technique [8], using a formal model of the required software behavior as the central component of our testing strategy [9], by using TestEra which means the framework for automated specification-based testing of Java programs [10]. TestEra automatically generates all nonisomorphic test inputs up to the given bound by using the method's pre-condition. The specifications are first-order logic formulas. Furthermore, there are studies to automatically generate test cases with an SMT solver. One is an SMT solving-based test generation approach for MATLAB Simulink designs, implemented in the HiLiTE tool was developed by Honeywell for the verification of avionic systems [11]. The other study is that testing-based formal verification with symbolic execution (TBFV-SE) checks whether programs correctly implement their formal specification [12]. They propose a method for automatically verifying theorems for TBFV-SE using SMT-solver. The related research mentioned here is test case generation from formal specifications or methods for automatic test case generation. Our tool achieves both of these objectives. Moreover, it does not only generate test cases but also generates test cases that are effective in detecting bugs in the program by using the vibration method technique.

Chapter 7

Conclusion and Future Work

In this paper, we describe how the original Vibration method can be improved to allow automatic test case generation from non-liner expressions and conjunctions of atomic predicates and also non-numeric data types, especially, set types, sequence types, and composite types. We also describe a supporting tool we have developed over the last three years for the Vibration method. The tool can automatically generate test cases from SOFL specifications, verify test results, and proving theorems. To evaluate our tool, we have conducted two experiments. The first experiment result shows that the accuracy is the same or better than that of traditional testing methods. In particular, we have shown that the Vibration method is more effective than other test case generation methods when the target program is a more detailed refinement of the specification. The second experiment proves that our support tool is capable of generating precise test cases from non-numeric test conditions based on an improved vibration method.

In spite of the progress we have made in the tool, it has limitations. One is that the tool cannot generate test cases from a conjunction with dependent variables or elements or fields. Furthermore, we still need a more systematic way to set distance for the Vibration method. Currently, the tool only supports random settings or manual settings of the distance, but its effectiveness is hard to be ensured. We will tackle these limitations in our future work.

Bibliography

[1] Shin Nakajima (2007) “Formal Methods as Software Engineering Tools,” NII

[2] Shaoying Liu, Shin Nakajima (2011) “A “Vibration” Method for Automatically Generating Test Cases Based on Formal Specifications,” 18th Asia-Pacific Software Engineering Conference, pp. 73-80

[3] Shaoying Liu (2004) “Formal Engineering for Industrial Software Development,” Springer-Verlag

[4] Goran Petrović, Marko Ivanković (2018) “State of Mutation Testing at Google,” 40th IEEE/ACM International Conference on Software Engineering, pp. 163-171

[5] A. Jefferson Offut, Shaoying Liu (1999) “Generationg test data from SOFL specifications,” The Journal of Systems and Software 49, pp. 49-62

[6] Shaoying Liu, A. Jeff Offut, Chris Ho-Stuart, Yong Sun, Mitsuru Ohba (1998) “SOFL: A Foraml Engineering Methodology for Industrial Applications,” IEEE Transactions on Software Engineering, 24(1), pp. 24-45

[7] Jonathan Peter Bowen, Kirill Bogdanov, John A. Clark, Mark Harman, Robert M. Hierons, Paul John Krause (2002) “FORTEST: Formal Methods and Testing,” 26th Annual International Computer Software and Applications Conference, pp. 91-101

[8] Jin, Hu, Zhi-shu Li, Qi Li (2006) “Specification-based Test Case Automatic Generation,” JOURNAL-SICHUAN UNIVERSITY NATURAL SCIENCE EDITION, 43(4), 772

[9] Heimdahl, Mats, Sanjai Rayadurgam, and Willem Visser (2001) “Specification centered testing.”

[10] Khurshid, S., Marinov, D. (2004) “TestEra: Specification-Based Testing of Java Programs Using SAT”. Automated Software Engineering 11, 403-434

[11] Ren Hao, Devesh Bhatt, and Jan Hvozdovic (2016) “Improving an Industrial Test Generation Tool Using SMT Solver,” NASA Formal Methods Symposium. Springer, Cham

[12] Sugai Kenta, Hiroshi Hosobe, and Shaoying Liu (2021) “SMT-Based Theorem Verification for Testing-Based Formal Verification,” 2021 10th International Conference on Software and Computer Applications

Publication List of the Author

[1] Kenya Saiki, Shaoying Liu, Hiroyuki Okamura, Tadashi Dohi (2021) “A Tool to Support Vibration Testing Method for Automatic Test Case Generation and Test Result Analysis,” The 21st IEEE International Conference on Software Quality, Reliability, and Security