Supporting Tool for Automatic Specification-Based Test Case Generation

Weihang Zhang[1,2] and Shaoying Liu[3]

[1] Graduate School of Software Engineering,
University of Science and Technology of China, Hefei, China
[2] Graduate School of Computer and Information Sciences,
Hosei University, Tokyo, Japan
Sea10494@mail.ustc.edu.cn
[3] Department of Computer and Information Sciences,
Hosei University, Tokyo, Japan
sliu@hosei.ac.jp

Abstract. Automatic test case generation is a potentially effective technique for program testing, but it still suffers from the lack of appropriate tool support. Our research presented in this paper mainly focuses on the developing of a tool for automatic test case generation based on formal specifications. We take advantage of the Liu's decompositional test case generation method and put forward a set of algorithms for automatically generating test cases based on various data types. A supporting tool on the application of the approaches is presented. The tool can generate test cases according to the users' given test conditions, and the result shows that our tool can produce test cases that satisfy most kinds of test conditions.

Keywords: automatic test case generation, specification, SOFL, decompositional method, functional scenario.

1 Introduction

Formal specification is one of the most important techniques of formal methods and it is used to precisely describe the most important information of the requirement for software systems. The target document of specification supported by our tool is the formal specification written in the SOFL, Structured Object-Oriented Formal Language [1]. It provides a practical method for developing software system and facilitating the subsequent development activities such as automatic test case generation and test result analysis.

Automatic test case generation based on formal specification is a potentially effective technique for software reliability. Several techniques are available for specification based test case generation. For instance, test case generated from algebraic specifications [2], from abstract state machines [3], and from B-method [4]. Liu et al. put forward a decompositional approach to automatic test case generation based on formal specifications [5]. The method is rigorous and practical, and it is good enough

S. Liu (Ed.): SOFL 2012, LNCS 7787, pp. 12–25, 2013.
© Springer-Verlag Berlin Heidelberg 2013

for realizing automation. However, there is no tool to support the entire automatic test case generation process. In this paper, we describe a supporting tool to support automatic test case generation based on SOFL specifications.

The structure of the supporting tool include generating test cases from various kinds of data types, such as Numeric Types, Character Types, String Types, Set Types, Sequence Types, and from compound predicate expressions, which include conjunction expressions and disjunction expression.

The remainder of this paper is organized as follows. Section 2 describes the concerned technique regarding the method of test case generation based on formal specification. Section 3 discusses the specific information on the design of the supporting tool. We will use a set of algorithms and some simple examples for illustration. Section 4 presents some details in a small experiment and introduces a prototype of the supporting tool. Section 5 introduces a brief overview of related work. Finally, we conclude the paper and point out future work in Section 6.

2 Approach to Automatic Specification-Based Test Case Generation

According to the work by Liu [5], the decompositional method of automatic test case generation based on formal specifications is concerned with generating a set of values that satisfy all the testing conditions. A testing condition of an operation specification is a constraint on the input variables and is expressed as predicate expression. With our method and the algorithms introduced in this paper, test case will be automatically derived from those predicate expressions.

In SOFL, the form of operation specification can be described as S(S_iv,S_ov)[S_pre,S_post], where S_iv denotes all input variables for the operation, S_ov represents all output variables whose values will be generated or updated after operation, and S_pre,S_post are the pre- and post-conditions of operation specification S, respectively.

1) Definitions: Suppose we have a post-condition of specification S, and it can be described as:

$$S_{post} = (C_1 \wedge D_1) \vee (C_2 \wedge D_2) \vee \ldots \vee (C_i \wedge D_i)$$

- Guard condition: A predicate Ci (i ∈ {1, 2…n}) is a "guard condition". The feature of guard condition is that it does not contain any output variables.
- Definition condition: A predicate Di(i ∈ {1,2,…n}) is a "definition condition", and there is at least one output variable but no guard condition.
- Functional scenario: In this case, a functional scenario f_S of S is a conjunction:Spre∧Ci∧Di.
- Functional scenario form (FSF): A disjunction expression $(S_{pre} \wedge C_1 \wedge D_1) \vee (S_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (S_{pre} \wedge C_i \wedge D_i)$ is a functional scenario form of specification S.

2) Testing condition: The testing condition in our method is the conjunction Spre∧Ci, where S_{pre} is the pre-condition, and C_i is the guard condition.

3) Test strategy: Suppose operation S has a FSF $(S_{pre} \wedge C_1 \wedge D_1) \vee (S_{pre} \wedge C_2 \wedge D_2) \vee \ldots \vee (S_{pre} \wedge C_i \wedge D_i)$. Let T be a test set for S. Then, T must satisfy the condition $(\forall_{i \in \{1,2,..n\}} \exists_{t \in T} S_{pre}(t) \wedge C_i(t))$

The test strategy means that every testing condition must be tested and its corresponding test case should be found in the test set T.

3 Supporting Tool for Automatic Test Case Generation Method

In our work, we aim to produce a package in C# to support automatic test case generation from various kinds of predicate expressions based on the component-based software engineering approach. Before describing the supporting tool, we will explain how a test case can be derived.

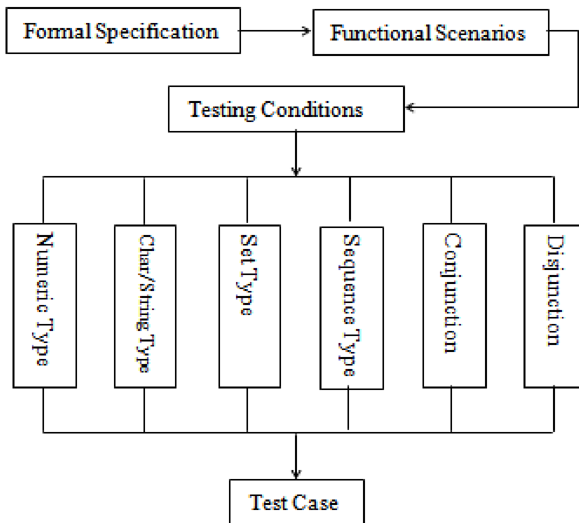

Fig. 1. Process of test case generation

Figure 1 shows the process of automatic test case generation based on the formal specification. As we have introduced in the previous sections, in order to generate test cases, functional scenarios derived from a formal specification must be given. And then, with the generated functional scenarios, we are able to obtain the testing conditions for the test. According to those derived test conditions, the supporting tool will be able to generate the corresponding test case in terms of different data types and expressions. In each chapter of this section, because the space of this paper is limited, the specific information about the data type and the introduction of their operators will be omitted. We just choose a few data types as examples and their corresponding algorithms, which can be used to generate test case based on different predicate expressions, for our discussion.

The algorithms for the operation of each data type are defined in each of the following classes, and all the classes are organized in the package named ASBTestCaseGeneration.

3.1 Test Case Generation Algorithms Based on Numeric Data Type

The algorithms are implemented using several methods in a class named Numeric. Each method deals with one specific case. The details of the methods are described below.

Method 1: GenerateFromSingleVar01 $(P(x^{\sim})\text{: } \mathbf{string}, \text{op: } \mathbf{string})$ x: **real**{…}. The input variable in this method is $x^{\sim}$, and the output variable is x.

Algorithm 1: We first discuss the algorithm for simple predicate expressions involving only one input variable, and any predicate expression $P(x^{\sim})$ can be transformed into the format as $x^{\sim}\Theta Exp$, where Θdenotes the relational operators of =, >, <, >=, <=, and <>, Exp is a constant expression which does not contain any variables. In such kind of situation, the algorithm for generating test cases is described below.

Suppose the predicate expression is expressed as the format of $x^{\sim}$ Θ Exp, then if Θ denotes =, we have x = Exp; And if Θ denotes >, then we have x = Exp + α, where α is a random positive numeric value. Also, if Θ denotes <, then we have x = Exp $-\alpha$, and α is a random positive numeric value as well. For the other situations, such as Θ denotes >=, <=, <>, the methods for generating test cases are the same as above, and they will not be described in detail.

Method 2: GenerateFromSingleVar02 $(P(x^{\sim})\text{: } \mathbf{string}, \text{opt:}\mathbf{string})$ x: **real**{…}.

Algorithm 2: Let us consider another format of simple predicate expression involving only one input variable, but the predicate expression $P(x^{\sim})$ is organized as the format Exp1Θ Exp2, where Exp1 and Exp2 are both arithmetic expressions, and they may contain variable $x^{\sim}$.

For this situation, the algorithm for generating test cases is described below.

Suppose the predicate expression has the format Exp1Θ Exp2, we should transform Exp1Θ Exp2 into the format $x^{\sim}\Theta Exp$, and then apply the Algorithm 1 to generate the value of x. For instance, suppose we have a predicate expression 2x +5>x+1, then we can transform this expression into the format of x > -4. Finally, after applying the algorithm 1, we can generate the value of x with -4+α, where α is a random positive numeric value.

Method 3: GenerateFromMultiVar ($P(x_1^{\sim}, x_2^{\sim}, \ldots, x_n^{\sim})\text{: } \mathbf{string}, \text{opt: } \mathbf{string})$ x_1, $x_2, \ldots,$ x_n: **real**{…}

Algorithm 3: The more complicated than the first two situations is when a predicate expression contains more than one input variables, and the predicate expression $P(x_1^{\sim}, x_2^{\sim}, \ldots, x_n^{\sim})$is expressed as the format Exp1Θ Exp2, where Exp1 and Exp2 are both arithmetic expressions, and they probably contain all the input variables

$x_1^{\sim}, x_2^{\sim}, \ldots, x_n^{\sim}$.The algorithm to process such kind of expression will be described below.

In order to generate test cases to satisfy $P(x_1^{\sim}, x_2^{\sim}, \ldots, x_n^{\sim})$,we should first make $x_1^{\sim}$ as the variable to be discussed in our algorithm, then randomly generate appropriate values for the input variables $x_2^{\sim}, x_3^{\sim}, \ldots, x_n^{\sim}$, respectively.

Eventually, we are able to derive the value of x_1 according to the method we have discussed for Algorithm 2.For example, suppose we have a predicate expression 3x+y+z > 2x +5. In order to automatically generate the values of x, y, z. Firstly, we should make x as the variable to be discussed in our method, and then randomly generate appropriate values for the input variables of y and z, such as y = 10, z = 20, therefore, the expression can be transformed into the format 3x+10+20 > 2x+5, which is suitable to apply the Algorithm 2 to generate the value of x. Finally, the test case satisfying $P(x_1^{\sim}, x_2^{\sim}, \ldots, x_n^{\sim})$ is: x=-25+α, y=10, z=20, where α is a random positive numeric value.

Method 4: GenerateFromLinearExp ($P(x_1^{\sim}, \ldots, x_n^{\sim})$: **string**, opt: **string**) $x_1, x_2, \ldots, x_n$: **real**{…}

Algorithm 4: In order to effectively and efficiently generate test cases, we use a special data structure to operate test case generation for linear equation. The data structure is described below:

Index	Variable	Real

With such kind of data structure, any linear equation such as ax + b could be expressed as the form as:

a	x	b

For example, 3x - 5 can be expressed as

3	x	-5

Since every leaf node in binary tree can be transformed into the particular structure described above, we are able to calculate linear equation easily.

Suppose any linear relational expression can be expressed as: exp1Θ exp2, where exp1 and exp2 both represent arithmetic expressions, Θ denotes the relational operators of =, >, <, >=, <=, and <>. Suppose we have exp1 = ax + m, exp2 = bx + n. And the expression is exp1 = exp2. Then, let us make exp1 the form we described above as:

a	x	m

exp2 as:

b	x	n

After calculating, we have derived another expression px + q = 0 and it can be described as:

p	x	q

Where p = a - b, q = m – n.

Finally, we can generate the value of x according to the expression x Θ (–b) / a, where Θ is a relational operator.

For example, we try to generate a value of x from the expression 4x+5 = 2x-2, where exp1 = 4x+5, exp2= 2x-2, and Θ represents =. Then, we have the structure for exp1:

4	x	5

And structure for exp2:

2	x	-2

After calculating the arithmetic expressions p=4-2=2, q= 5-(-2)=7, we have another structure for the result:

2	x	-7

Eventually, a value we generated is x = (-7)/2 = -3.5

Method 5: GenerateFromQuaExp ($P(x_1{}^{2\sim}, x_2{}^{\sim}, \ldots, x_n{}^{\sim})$: **string**, opt: **string**) $x_1, x_2, \ldots, x_n$: **real**{…}

Algorithm 5: For quadratic equations:

As you can see, we are able to use this kind of structure to describe any kinds of

Index-a	Index-b	Index-c	Variable

quadratic equations such as ax^2+bx+c, and then we can get the corresponding structure as

a	b	c	x

For example, expression x^2+2x+4 can be described as the following form,

1	2	4	x

Since the method of transforming quadratic expression into the particular structure is similar to the linear expression, we can easily describe the specific structures for the quadratic equation of x^2+2x-2=1.

Then, the expression x^2+2x-2 can be described below,

1	2	-2	x

Accordingly, the value 1 will be described as

0	0	1	Null

After calculating, we have derived another expression x^2+2x-3=0. And it can be transformed into the structure as below,

1	2	-3	x

As we know, for quadratic equation, when $b^2 - 4ac \geq 0$, we can generate the values of x from the expression $x = \frac{-b\mp\sqrt{b^2-4ac}}{2a}$, and if $b^2 - 4ac < 0$, then we cannot get the value of x.

Here, we know that a=1, b=2, c=−3, and $2^2 - 4*1*(-3) = 16 > 0$, so we can generate the values of x where $x_1 = \frac{-2+\sqrt{2^2-4*1*(-3)}}{2*1} = 1, x_2 = \frac{-2-\sqrt{2^2-4*1*(-3)}}{2*1} = -3.$

Finally, we can generate the test case from the quadratic equation that is $x_1 = 1$ and $x_2 = -3$.

Since the space of this paper is limited, we will not give the corresponding structure of binary tree in detail.

3.2 Test Case Generation Algorithms Based on Set Type

In this chapter, we focus our discussion on the algorithms of automatically deriving test cases from an expression involving all the input variables of the set type operator. Since the underlying principles of the algorithms for all the set type operators in SOFL are similar and the space of this paper is limited, we only choose some operators as examples for our discussion.

The algorithms are implemented using several methods in a class named Set. Each method deals with one specific case. The details of the methods are described below.

Method 6: GenerateFromSubset(x_1~: **set**)x: **set**, x_1: **set**{…}

Algorithm 6: Let us first use a simple example to explain the algorithm for the operator subset. Consider the predicate expression x subset x_1~. To generate a test case to satisfy this expression, according to the method introduced in Algorithm 3, we first randomly produce a set value for variable x_1~, and then in order to generate a test case, we just need to appropriately produce the values of x. We can take any elements in the generated set x_1 to make a new set value. Finally, the values of x_1 and x will satisfy the predicate expression, and they are the results of our test.

For example, suppose we want to generate a test case from the expression x subset x_1. Firstly, according the method, x_1 will be randomly generated, suppose it is {4,9,12}. And then, to decide the value of x, we just need to get some elements from the set x_1 we produced just now, suppose x is {9,12}. Finally, a test case for our test is x = {9, 12} and x_1={4,9,12}.

Additionally, for the expression x_1 union(x_2 inter x_3) subset x_4 uion x_5, where variables x_1, x_2,x_3,x_4,x_5 are all input variables of the set type, it is a compound expression involving different operators, and it will be discussed in subsequence sections.

Method 7: GenerateFromUnion(x_1~: **set**, x_2~: **set**)x: **set**, x_1: **set**, x_2: **set**{…}

Algorithm 7: Let us consider anther algorithm for the operator union. Suppose we have an expression x=union (x_1~, x_2~), wherex_1~, and x_2~ are all input variables of the operator union. To generate a test case for this predicate expression, we should also first randomly produce set values for variables x_1~, x_2~, and then it is quite simple to derive the result of the operation union (x_1, x_2). We can obtain all the elements of x_1 in the resulting set x and then add the members of x_2 that are not contained in x_1. Finally, the generated set values of x, x_1, and x_2 that satisfy the predicate expression are the test case for our test.

For example, suppose we have an expression x=union (x_1, x_2), to generate a test case from this expression, according to the algorithm, we should first randomly generate the values for the sets x_1 and x_2, suppose x_1 = {15, 17, 18, 20, 22} and x_2 = {8, 9, 17, 20, 23}. Then, obtain all the elements of x_1 to the set x, x = {15, 17, 18, 20, 22}. We can produce a suitable value for set x by adding the members of x_2 that

are not contained in x_1, so x will be {15, 17, 18, 20, 22, 8, 9, 23}. Finally, a test case for our test is x = {15, 17, 18, 20, 22, 8, 9, 23}, x_1 = {15, 17, 18, 20, 22} and x_2 = {8, 9, 17, 20, 23}.

Method8: GenerateFromInter(x_1~: **set**, x_2~: **set**)x: **set**, x_1: **set**, x_2: **set**{... }

Algorithm 8: Let us discuss the algorithm for the operator inter. Let x_1~, x_2~ are all input variables of the operator inter, and x=inter (x_1, x_2) is the target predicate expression. In order to generate a test case to satisfy the expression, the method is very similar to Algorithm 6 introduced above. Firstly, the set values for variablesx_1~,x_2~ will be randomly produced, and then we focus on how to generate values for variable x, we will give a pseudo code to explain this method:

```
Set Inter(Set s1, Set s2){
   Set result;
   for (i: =0 to s1.length - 1){
      k: = 0;
      while (s1 [i] != s2[k] && k< s2.length){
         k++;
      }
      if (k >= s2.length)
         i++;
      else{
         add s1 [i] to the set result;
         i++;
      }
   }
   return result;
}
```

Finally, we will obtain the result set value that represents the test case for variable x, and with the generated value of x_1, and x_2, we have successfully gained the test case for all input variables of the target predicate expression.

3.3 Test Case Generation Algorithms Based on Sequence Type

In this section, we will move forward to discuss the algorithms for automatically deriving test cases from a predicate expression involving all the input variables of the sequence type operator. As we mentioned above, because the underlying principles of algorithms for the operators in sequence type are quite similar, we just choose some operators (subsequence, elements and concatenation) as examples for our discussion, without giving all the descriptions for every operator in detail.

The methods for processing the sequence type are defined in a class named Sequence.

Mehtod 9: GenerateFromSubseq(S: **seq**, i: **int**,j: **int**) x: seq{…}

Algorithm 9: In this part, we will describe the algorithm for the operator subsequence.

Let S,i and j be input sequence variables, consider the predicate expression x = S (i, j), where i and j are both integer values, and the expression means obtaining the elements in sequence S from the position i to the position j, then make the obtained elements as a new sequence that is the subsequence of sequence S.

To generate a test case to satisfy this expression, according to the method introduced in Algorithm 3, we first randomly produce a sequence value for variable S, and the length of sequence S must be not less than j, and then in order to generate a test case for x, we just need to get values from the generated sequence S from the position i to the position j. The elements we got from sequence S will be added into the sequence of x. Finally, the values of S and x will satisfy the predicate expression, and they are the results of our test.

Method 10: GenerateFromElems(x_1: **seq**)x: set{…}

Algorithm 10: Let us discuss the algorithm for the operator elems. Let x_1, xbe the input and output variables of the operator elems, respectively. And x_1 is sequence type, x=elems(x_1) is the target predicate expression. To generate a test case for this predicate expression, we should also first randomly produce a sequence value for variables x_1 and then it is quite simple to derive the result of the operation x=elems(x_1). We can obtain all the elements from the sequence x_1, and then add the members to the set x to form a new set value. Finally, the generated set value of x, and sequence value of x_1 that satisfy the predicate expression are the test cases for our test.

Method 11: GenerateFromConc(x_1: **seq**,x_2: **seq**) x: seq{…}

Algorithm 11: Let us consider anther algorithm for the operator conc. Suppose we have an expression x=conc(x_1, x_2) , where x_1, and x_2 are all input variables of the operator conc. In order to generate a test case to satisfy the expression, the method is very similar to Algorithm 7 introduced above. The only difference is that in sequence, the duplication values are allowed to appear in a same sequence. Therefore, it is quite simple to generate test case for this operator. Firstly, we should randomly produce the sequence values for variables x_1, x_2, after that we include all the members of the generated sequence x_1 in the sequence x and then extend it by adding the members of the generated sequence x_2.

Finally, with the generated value of x, x_1 and x_2, we have successfully gained the test case for all input variables of the target predicate expression.

For example, in order to generate a test case from the expression x=conc(x_1, x_2), we should first randomly produce the values for variables x_1 and x_2, suppose x_1 = [1,2,3,4,5] and x_2 = [4,5,6,7,8]. Then, we are able to obtain the value for variable x by combining two sequence values of x_1 and x_2. Finally, x = [1, 2, 3, 4, 5, 4, 5, 6, 7, 8], x_1 = [1,2,3,4,5] and x_2 = [4,5,6,7,8] are the test cases for our test.

3.4 Algorithms for Automatic Test Case Generation Based on Conjunction and Disjunction Expressions

We have introduced each basic data type, and two compound data types of Set and Sequence in the previous sections. In this section, we will introduce the Conjunction expression and the Disjunction expression, respectively. In each compound predicate expression, no matter Conjunction or Disjunction, it will probably involve compound data types (e.g., numeric, string, set, sequence), which are introduced in the previous

sections. We will introduce the algorithms in detail on how to generate test cases according to those kinds of compound predicate expressions.

1) For Conjunction Predicate Expressions: We will describe the algorithm for conjunction expression in this section. The methods for dealing with the conjunction expression are defined in a class named Conjunction.

Method 12: GenerateFromConjunctionExp(exp: **string**) $x_1, x_2, \ldots, x_n$: **real** {…}

Algorithm 12: To generate a test case for conjunction, the test case must satisfy all the atomic predicate expressions in the conjunction. The fundamental idea for test case generation for a conjunction is that we should first generate a group of values for all the input variables of the operation from one of atomic predicate expression using the algorithms introduced in the previous sections. And then we test the values to make sure whether they satisfy other atomic predicate expressions or not, we will find a test case for the conjunction if the test case satisfy all the remaining conjunction constituents; Otherwise, it means the values are not suitable for the conjunction, and we should use the algorithm again to generate another test value, and repeat the above procedure until we find a suitable test case for all the constituents of the conjunction predicate expression.

In order to explain the main idea of the algorithm explicitly, a pseudo code will be given below:

```
voidGenerateFromConjunctionExp(String exp)
//The formal parameter exp is the target conjunction pre-
dicate expression, and the format of the expression
is Q_i^1∧Q_i^2∧…∧Q_i^w. Each atomic expression and each operator in
the atomic expression can be analyzed and detected, using
the function ConstAnalyse(string exp); However, since the
space in the paper is limited, the specific algorithm for
this function will be omitted.
{
   j:= 1;
   successful:= true;
   ConstAnalyse(string exp) { … } //Using this function to
analyze the expression, it will return a resulting  list
which contains every separate atomic expression in the
conjunction predicate expression Q_i^1∧Q_i^2∧…∧Q_i^w.
   GenerateFromAtomicExp(string aExp) // Generate r values
v1,v2, … , vr as a test case that satisfies Q_i^j, aExp is a
value from the list generated from function ConstAna-
lyse(string exp).
   j:=j+1;
   while(j<=w && j <const)   // const is a given number,
in order to control the amount of Loop to avoid dead lock
happening in the program.
   {
      if (Q_i^j (v1,v2,…,vr))
```

```
        j=j+1; // Q_i^j (v1,v2,…,vr)  means whether values
v1,v2, … ,vr satisfying the atomic expressionQ_i^jor not, if
it returns true, it says that the derived values satisfy
the expression Q_i^j, otherwise, we have to generate the val-
ues again.
      else {
         successful = false;
         break;
      }
   }
   if (successful=true)
      Output the r values v1,v2,…, vr as a successful test
case;
   else
      Output a message "no test case is generated.";
}
```

2) For Disjunction Predicate Expressions: We will describe the algorithm for disjunction predicate expression. The methods for dealing with the disjunction expression are defined in a class named Disjunction.

Method 13: GenerateFromDisjunctionExp(exp: **string**) x_1, x_2, …, x_n: **real** {…}

Algorithm 13: Compared with conjunction predicate expressions, test case generation from a disjunction seems much simpler. To generate test cases for the disjunction $Q_1 \vee Q_2 \vee \ldots \vee Q_m$, we just have to generate one test case for each disjunction constituent until all the atomic predicate expressions in the disjunction are covered, respectively. Finally, the generated test cases constitute a complete test set that is the result for the disjunction.

An algorithm of automatic test case generation from $Q_1 \vee Q_2 \vee \ldots \vee Q_m$ will be given below:

```
GenerateFromDisjunctionExp(String str) {
   j:= 1;
   ConstAnalyse(string exp) { … } //Using this function to
analyze the expression, it will return a resulting  list
which contains every separate atomic expression in the
conjunction predicate expression Q1VQ2V…VQm.
   while(j<=m)
   {
      Generate and output r values v1,v2,…,vr as a suc-
cessful test case that satisfies Qj using the algorithm
given in the previous sections.
      j:= j+1;
   }
   return;
}
```

4 Design of the Tool

In this section, we will briefly introduce the prototype tool for supporting the automatic specification-based test case generation methods. The support tool is implemented using Visual Studio .Net 2010 with language C#.

In order to explain our work clearly and help readers understand the techniques for the specification-base test case generation, we use a very simple case for illustration. According to the work done by Liu et al. [8], we assume that in our tool, the function of automatically generating all the test conditions from the derived functional scenarios of an operation, which are automatically generated from formal specification, have been realized. An example for the implementation of the tool will be given below.

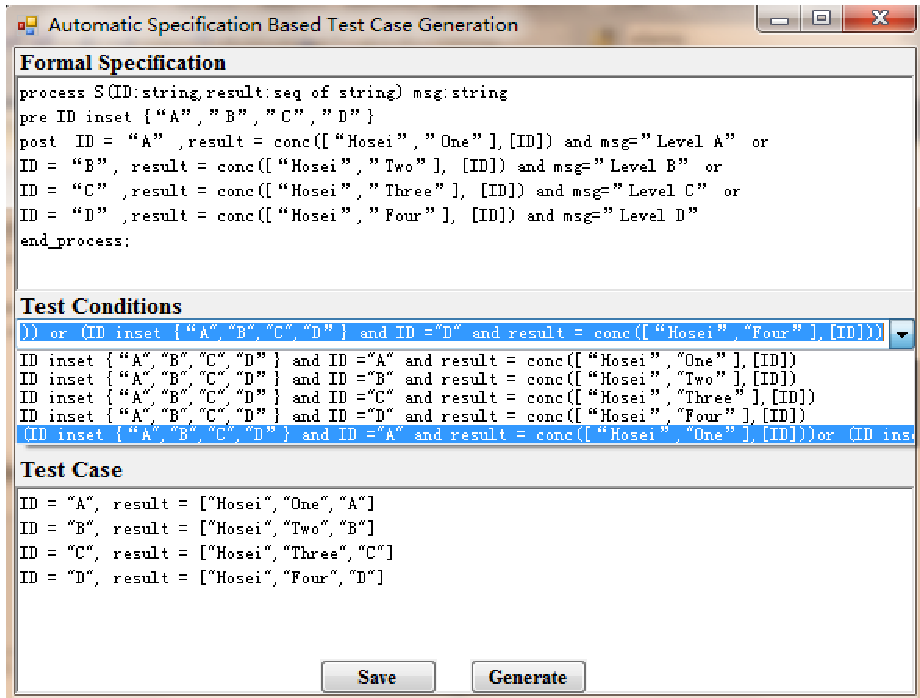

Fig. 2. Illustration of set type predicate expression

Figure 2 shows an illustration of processing compound predicate expressions, as we can see in the picture, test conditions for this process is ID inset {“A”,”B”,”C”,”D”} and (ID = “A” ,result = conc([“Hosei”,”One”],[ID])) or (ID = “B”, result = conc([“Hosei”,”Two”], [ID])) or (ID = “C” ,result = conc([“Hosei”,”Three”], [ID])) or (ID = “D” ,result = conc([“Hosei”,”Four”], [ID])). The test conditions are associated with the Set and Sequence types, therefore, we should use the method for dealing with compound expressions. Eventually, the corresponding test case will be derived after processing.

5 Related Work

The decompositional approach to automatic test case generation based on formal specification was first introduced in Liu's paper [5], and it serves as a fundamental principle of the design of our supporting tool. Since the method just describe the main idea of automatic test case generation, in this paper, we have discussed some explicit algorithms. In additional, there have existed various methods for specification-based test case generation based on various specification techniques.

Bandyopadhyay et al.[3] put forward a testing methodology that combines information from UML sequence models and state machine models into one testable model based on the improvement of the work of Dinh-Trong et.al., which provided an approach to combine information from a class and a sequence diagram to generate test input. Based on state machine models, they use a testing method to select a set of transition sequences according to state machine coverage criteria, and then, with those generated transition sequences, the tool they built to support their approach is able to generate test inputs for each transition sequence.

Khrushid el al. [6] built a framework called TestEra for automatic testing of Java program based on specification. Their tool employs Alloy analyzer to produce instance of Alloy specification, where Alloy is a first-order declarative language based on sets and relations. After that, using the pre- condition of those generated instance of Alloy specification, the tool can automatically generate all non-isomorphic test inputs. Furthermore, TestEra can automatically generate the corresponding Jave data structure according to the description of the structural invariants of inputs.

Simon Burton [7] presents a framework of automatically generating tests for Z specification based on user-defined test criteria. Heuristics can be used to detect errors with the given resource constraints of the process. The framework allows for the automatic and formally generation of test sets based on formally defined testing heuristics. In the tool, test cases can be automatically generated by formalizing testing heuristics, analyzing properties of these heuristics.

6 Conclusion and Future Work

We have described the design and implementation of a supporting tool for automatic test case generation based on formal specifications. Formal specification in terms of pre- and post- conditions has tremendous advantages to be effectively utilized to generate test cases for testing programs. And tool support is crucial for the application of automatic test case generation approach based on formal specification. Our tool presented in this paper provides a package including many classes. Each class is designed to process each data type, respectively. Correspondingly, there are a lot of algorithms defined in each class for automatically generating test case according to different operators and predicate expressions. Our supporting tool is also crucial for the further research of automatic software testing. For example, our tool can serve as the foundation for testing result analysis, each component of this tool can be reused and integrated with the tool of testing result analysis easily.

In the future, we plan to make our further research to develop a set of more efficient algorithms for automatic test case generation. Since there are still some challenges in automatic testing, for example, it is difficult to deal with some set expressions including infinite set, such as x inset S, where S is a very large or infinite set. Therefore, in order to totally realize automatic testing, our future work should be focused on the algorithms that are capable to deal with all kinds of complicated expressions with a practically acceptable efficiency.

References

1. Liu, S.: Formal Engineering for Industrial Software Development Using the SOFL Method. Springer (2004) ISBN 3-540-20602-7
2. Gaudel, M.-C., Le Gall, P.: Testing Data Types Implementations from Algebraic Specifications. In: Hierons, R.M., Bowen, J.P., Harman, M. (eds.) FORTEST. LNCS, vol. 4949, pp. 209–239. Springer, Heidelberg (2008)
3. Bandyopadhyay, A., Ghosh, S.: Test Input Generation using UML Sequence and State Machines Models. In: Proceedings of 2nd International Conference on Software Testing, Verification, and Validation (ICST), Denver, USA, April 1-4, pp. 121–130. IEEE CS Press (2009)
4. Utting, M., Legeard, B.: Practical Model-Based Testing: A Tools Approach. Morgan Kaufmann (2007)
5. Liu, S., Nakajima, S.: Decompositional test case generation based-on specification. In: 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement, June 09-11 (2010)
6. Khurshid, S., Marinov, D.: TestEra: Specication-based Testing of Java Programs using SAT. Automated Software Engineering 11(4) (2004)
7. Burton, S.: Automated Testing from Z Specifications, TR YCS-2000-329, University of York, UK (2000)
8. Liu, S., Hayashi, T., Takahashi, K., Kimura, K., Nakayama, T., Nakajima, S.: Automatic Transformation from Formal Specifications to Functional Scenario Forms for Automatic Test Case Generation. In: 9th International Conference on Software Methodologies, Tools, and Techniques, Yokohama, Japan, September 29-October 1, pp. 383–397. IOS Press (2010)