

Supporting Tool for Automatic Specification-Based Test Case Generation

ZHANG, Weihang

(出版者 / Publisher)

法政大学大学院情報科学研究科

(雑誌名 / Journal or Publication Title)

法政大学大学院紀要. 情報科学研究科編 / 法政大学大学院紀要. 情報科学研究科編

(巻 / Volume)

8

(開始ページ / Start Page)

37

(終了ページ / End Page)

42

(発行年 / Year)

2013-03

(URL)

<https://doi.org/10.15002/00009541>

Supporting Tool for Automatic Specification-Based Test Case Generation

WeiHang Zhang
Graduate School of Computer and Information Sciences
Hosei University
E-mail: weihang.zang.2b@stu.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

I. 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 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 II describes the concerned technique regarding the method of

test case generation based on formal specification. Section III 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 V introduces a brief overview of related work. We conclude the paper and point out future work in Section IV.

II. 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:

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

· *Guard condition:* A predicate C_i ($i \in \{1, 2, \dots, n\}$) is a "guard condition". The feature of guard condition is that it does not contain any output variables.

· *Definition condition:* A predicate D_i ($i \in \{1, 2, \dots, n\}$) 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: $S_{pre} \wedge C_i \wedge D_i$.

· *Functional scenario form (FSF):* A disjunction expression $(S_{pre} \wedge C_1 \wedge D_1) \vee (S_{pre} \wedge C_2 \wedge D_2) \vee \dots \vee (S_{pre} \wedge C_i \wedge D_i)$ is a functional scenario form of specification S .

2) *Testing condition*: The testing condition in our method is the conjunction $S_{pre} \wedge C_i$, where S_{pre} is the pre-condition, and C_i is the guard condition.

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

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

III. 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.

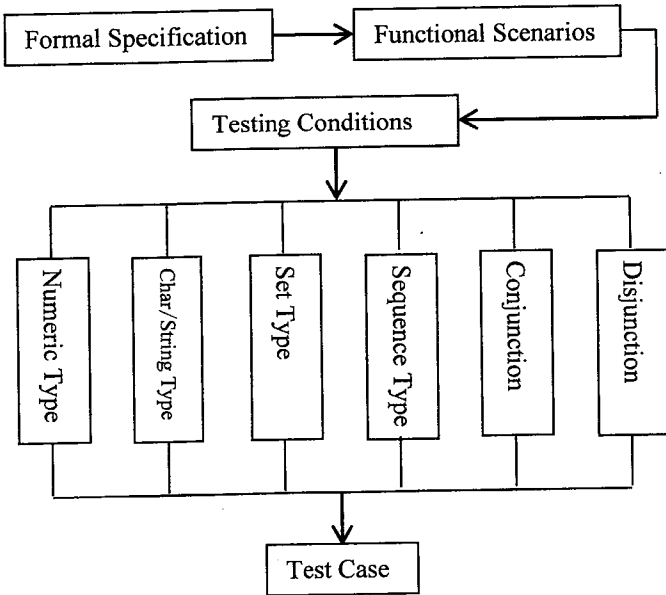


Figure1. Process of test case generation

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.

A. 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 ($P(x^{\sim})$: **string**, opt: **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 $x^{\sim} \Theta \text{Exp}$, where Θ denotes the relational operators of $=, >, <, \geq, \leq,$ and \diamond , 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 $x^{\sim} \Theta \text{Exp}$, then if Θ denotes $=$, we have $x = \text{Exp}$; And if Θ denotes $>$, then we have $x = \text{Exp} + \alpha$, where α is a random positive numeric value. Also, if Θ denotes $<$, then we have $x = \text{Exp} - \alpha$, and α is a random positive numeric value as well. For the other situations, such as Θ denotes \geq, \leq, \diamond , the methods for generating test cases are the same as above, and they will not be described in detail.

Method 2: GenerateFromSingleVar02 ($P(x^{\sim})$: **string**, opt: **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 $\text{Exp1} \Theta \text{Exp2}$, 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 $\text{Exp1} \Theta \text{Exp2}$, we should transform $\text{Exp1} \Theta \text{Exp2}$ into the format $x^{\sim} \Theta \text{Exp}$, 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 $-4 + \alpha$, where α is a random positive numeric value.

Method 3: GenerateFromMultiVar ($P(x_1^{\sim}, x_2^{\sim}, \dots, x_n^{\sim})$: **string**, opt: **string**) x_1, x_2, \dots, 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 $P(x_1^{\sim}, x_2^{\sim}, \dots, x_n^{\sim})$ is expressed as the format $\text{Exp1} \Theta \text{Exp2}$, where Exp1 and Exp2 are both arithmetic expressions, and they probably contain all the input variables $x_1^{\sim}, x_2^{\sim}, \dots, x_n^{\sim}$. The algorithm to process such kind of expression will be described below.

In order to generate test cases to satisfy $P(x_1, x_2, \dots, x_n)$, we should first make x_1 as the variable to be discussed in our algorithm, then randomly generate appropriate values for the input variables x_2, x_3, \dots, x_n , 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 $P(x_1, x_2, \dots, x_n)$ is: $x = -25 + \alpha, y = 10, z = 20$, where α is a random positive numeric value.

Method 4: GenerateFromLinearExp ($P(x_1, x_2, \dots, x_n)$)

: **string**, opt: **string**) x_1, x_2, \dots, 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: $\text{exp1} \Theta \text{exp2}$, where exp1 and exp2 both represent arithmetic expressions, Θ denotes the relational operators of $=, >, <, \geq, \leq,$ and \diamond . Suppose we have $\text{exp1} = ax + m, \text{exp2} = bx + n$. And the expression is $\text{exp1} = \text{exp2}$. 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 $x \Theta (-b) / a$, where Θ is a relational operator.

For example, we try to generate a value of x from the expression $4x + 5 = 2x - 2$, where $\text{exp1} = 4x + 5, \text{exp2} = 2x - 2$, 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 $x = (-7)/2 = -3.5$

B. 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 5: GenerateFromSubset(x_1 : **set**) x : **set**, x_1 : **set**{...}

Algorithm 5: Let us first use a simple example to explain the algorithm for the operator subset. Consider the predicate expression $x \text{ subset } x_1$. 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 $x \text{ subset } x_1$. 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 $x_1 \text{ union } (x_2 \text{ inter } x_3) \text{ subset } x_4 \text{ union } x_5$, 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 6: GenerateFromUnion(x_1 : **set**, x_2 : **set**) x : **set**, x_1 : **set**, x_2 : **set** {...}

Algorithm 6: Let us consider another algorithm for the operator union. Suppose we have an expression $x = \text{union}(x_1, x_2)$, where x_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 and 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 $x = \text{union}(x_1, x_2)$, 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 $x_2 = \{8, 9, 17, 20, 23\}$.

Method7: `GenerateFromInter(x1: set, x2: set)`
`x: set, x1: set, x2: set { ... }`

Algorithm 7: Let us discuss the algorithm for the operator inter. Let x_1 and x_2 are all input variables of the operator inter, and $x = \text{inter}(x_1, x_2)$ 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 variables x_1 and 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.

C. 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.

Method8: `GenerateFromSubseq(S: seq, i: int, j: int)`

`x: seq{ ... }`

Algorithm 8: 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.

Method9: `GenerateFromElems(x1: seq) x: set{ ... }`

Algorithm 9: Let us discuss the algorithm for the operator elems. Let x_1 and x be the input and output variables of the operator elems, respectively. And x_1 is sequence type, $x = \text{elems}(x_1)$ 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 $x = \text{elems}(x_1)$. 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.

Method10: `GenerateFromConc(x1: seq, x2: seq)`
`x: seq{ ... }`

Algorithm 10: Let us consider another algorithm for the operator conc. Suppose we have an expression $x = \text{conc}(x_1, x_2)$, 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 6 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 $x = \text{conc}(x_1, x_2)$, 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.

D. Algorithms for automatic test cases generation based on Conjunction and Disjunction 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 11: GenerateFromConjunctionExp(exp: string)

x_1, x_2, \dots, x_n : real {...}

Algorithm 11: 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:

```
void GenerateFromConjunctionExp(String exp)
```

//The formal parameter exp is the target conjunction predicate expression, and the format of the expression is $Q_i^1 \wedge Q_i^2 \wedge \dots \wedge Q_i^w$. 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  $Q_i^1 \wedge Q_i^2 \wedge \dots \wedge Q_i^w$ .
```

```
    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 ConstAnalyse(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 ( $Q_i^j(v1,v2,\dots,vr)$ ) j=j+1; //  $Q_i^j(v1,v2,\dots,vr)$  means
    whether values v1,v2, ... ,vr satisfying the atomic expression
     $Q_i^j$  or not, if it returns true, it says that the derived values satisfy
    the expression  $Q_i^j$ , otherwise, we have to generate the values
    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 12: GenerateFromDisjunctionExp(exp: string)

x_1, x_2, \dots, x_n : real {...}

Algorithm 12: Compared with conjunction predicate expressions, test case generation from a disjunction seems much simpler. To generate test cases for the disjunction $Q_1 \vee Q_2 \vee \dots \vee Q_m$, 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 $Q_1 \vee Q_2 \vee \dots \vee Q_m$ 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 $Q_1 \vee Q_2 \vee \dots \vee Q_m$.

```

while(j<=m)
{
Generate and output r values v1,v2,...,vr as a successful test
case that satisfies  $Q_j$  using the algorithm given in the previous
sections.
    j:=j+1;
}
return;
}

```

IV. 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 addition, 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 Java 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.

V. CONCLUSIONS 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, if a conjunction constituent is a complicated multiple equation, such as $x^3 + \sqrt[3]{y} + z^5 > m^{15} + y^3$, where all the relevant variables in the expression are input variables. It is extremely difficult for us to automatically generate test case for this equation using the current algorithms. Similarly, it is also difficult to deal with some set expressions including infinite set, such as $x \in S$, 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] Shaoying Liu, "Formal Engineering for Industrial Software Development Using the SOFL Method," Springer-Verlag, ISBN 3-540-20602-7, 2004.
- [2] M. C. Gaudel and P. Le Gall, "Testing Data Types Implementation from Algebraic Specifications," In R. Hierons, J. Bowen, and M. Harman, editors, Formal Methods and Testing, pages 209-239. LNCS 4949, Springer-Verlag, 2008.
- [3] A. Bandyopadhyay and S. Ghosh, "Test Input Generation using UML Sequence and State Machines Models," In Proceedings of 2nd International Conference on Software Testing, Verification, and Validation (ICST), pages 121-130, Denver, USA, April 1-4 2009. IEEE CS Press.
- [4] M. Utting and B. Legeard, "Practical Model-Based Testing: a Tools Approach," Morgan Kaufmann, 2007.
- [5] Shaoying Liu and Shin Nakajima, "Decompositional test case generation based-on specification," 2010 Fourth International Conference on Secure Software Integration and Reliability Improvement. June 09-June 11, 2010.
- [6] S. Khrushid and D. Marinov. "TestEra: Specication-based Testing of Java Programs using SAT," Automated Software Engineering, 11(4), 2004.
- [7] S. Burton. "Automated Testing from Z Specifications," TR YCS-2000-329, University of York, UK, 2000.