A Study for Extended Regular Expression-based Testing*

Pan Liu[1,2], Jun Ai[3], Zhenning(Jimmy) Xu[4]

[1] *College of Information and Computer, Shanghai Business School, Shanghai 201400, China* [2] *Shanghai Key Laboratory of Computer Software Testing and Evaluating, Shanghai, China* [3] *School of Reliability and System Engineering, Beihang University, Beijing, China*

[4] *School of business, University of Southern Maine, USA*

panl008@163.com, aijun@buaa.edu.cn, zhenning.xu@maine.edu

Abstract

Software testing has become an essential activity to guarantee software quality. To reduce the overall cost of software testing, model-based testing has been widely studied in the past two decades and Finite State Machine (FSM) is used to build the model of software behaviors. However, due to the inadequacy of the modeling ability of FSM, FSM-based testing cannot be taken as a test oracle to solve all issues in software testing. To improve the modeling capability of the model, a few researchers have proposed using Extended Regular Expressions (ERE) to model software behaviors. This paper reviews the method of the ERE-based testing and presents six modeling rules to convert program codes to the ERE model. Then, a case is adopted to illustrate the process of generating executable paths from the ERE model and the method of designing test cases by those paths. Compared with the traditional graphic traversal method of constructing executable paths from the program, ERE not only has robust modeling capability to describe more types of software behaviors than FSM, but also can be used to construct effective executable paths to detect program errors.

Keywords- software testing; model-based testing; ERE-based testing; extended regular expression.

1. Introduction

Software has been the information infrastructure, penetrated into all fields of economy, military and personal life. However, due to the lack of software testing technology, software quality is not still satisfactory, resulting in the problem of software trust [1-3]. For example, in 2009, DNS failure of Storm Video Website led to the collapse of internet network in some provinces of southern China. To detect software errors, software testing, as a measure for ensuring software equality, has been adopted in the software development lifecycle. However, it was reported that the traditional manual testing method has occupied more than 50% of the total cost of software development [4]. To decrease the cost of software testing, some automated test methods have been highlighted in the software engineering field.

Model-based testing [5-7], an automated test method, can generate test cases from the model describing system requirements, and then detect software errors by observing the inconsistency between executing results of the software and the desired model. Since this test method allows testers to evaluate requirements independent of algorithm design and development, it realizes “Test-First programming” and makes testers test software at each stage of the software lifecycle.

Nevertheless, model-based testing is not yet available as a test oracle [4]. An important reason is that the modeling ability of the traditional model (such as FSM) [5, 8] is far from perfect. For instance, FSM is not appropriate for describing the different types of cycles in software behaviors. In the past, some scholars were used regular expressions to model software behaviors and generate test cases [9, 10]. Since regular expressions include two specific notations “*” and “+” denoting two types of cycles in software behaviors, they are more suitable for modeling software behaviors than FSM.

To improve the modeling ability of regular expressions, we have put forward a theory of test

* This work is supported by National Natural Science Foundation of China (NSFC) under grant (No. 61502299), Science and technology key project of Jiangxi Province (No. 20142BBE50015).

modeling based on extended regular expressions (ERE) [8]. Compared with FSM, ERE has more powerful modeling ability to describe more types of cycles in software behaviors. In this paper, we presented six modeling rules of constructing the ERE model for a program. Then, through a case, we studied the modeling method in terms of these modeling rules and the test generation process by using the formal method. Different from model-based testing, ERE-based testing can generate effective executable paths without traversing the graph (FSM).

This paper is organized as follows: Section 2 introduces the modeling theory of extended regular expressions. Section 3 presents six modeling rules to build the ERE model of the program. Section 4 gives a case to study the modeling method of ERE and the process of test generation from the ERE model. Section 4 discusses the difference between the ERE-based testing method and the FSM-based method. Section 5 concludes the whole paper and points out future directions.

2. Preliminaries

In this section, we will introduce some basic conceptions in ERE-based testing, and discuss the relationship between ERE and the System Under Testing (SUT).

Definition 1 (Transition Set): A transition set is defined as $TN \subseteq S \times I/O \times S$, where

- S is the set of states,
- I is the set of input conditions on the states, and
- O is the set of the output results on the states.

By definition 1, any transition can be denoted as $tr=(s_1, i/o, s_2)$, where s_1 is called the pre-state of tr, s_2 is called the next-state of tr, i denotes the input condition on s_1, and o denotes the output result on s_2. We use the notation "-" to describe that there is not an input condition on s_1 or an output on s_2 in tr. To simplify the description of transitions in the model, in this paper, we adopt an alphabet to represent a transition and then use transitions to construct the ERE model.

Definition 2(Empty Transition): The empty transition is denoted by the notation ε, which indicates that a transition does not be executed.

In regular expressions, the empty transition ε plays an important role in the operation of regular expressions. For example, $a^0 = \varepsilon$ indicates that the transition a does not be executed in the SUT.

Definition 3 (Concatenation Operator): The concatenation operator is denoted by the notation ".", which indicates a concatenation operation between two transitions.

By definition 3, $a.b$ denotes that the transition a occurs first, and then the transition b occurs in the SUT. Using concatenation operator, we can get the relationship between a regular expression and transitions. For example, a regular expression $k^3=k.k.k$ denotes the concatenation of three transitions k. And an equation $k.\varepsilon=\varepsilon.k=k$ exists for any transition k. Let Σ be a nonempty set of transition sequences. There are some properties for concatenation operator as follows [8]:

1) $\forall a,b \in \Sigma \bullet a.b \neq b.a \Rightarrow a \neq b \wedge a \neq \varepsilon \wedge b \neq \varepsilon$
2) $\forall a,b,c \in \Sigma \bullet a.b.c = (a.b).c = a.(b.c)$
3) $\forall a \in \Sigma \bullet a.\varepsilon = \varepsilon.a = a$

Definition 4 (Transition Sequence): A transition sequence ts is a set of concatenation operations of transitions.

Any transition sequence ts denotes a path fragment of the SUT. So, if we can find a transition sequence, starting from the initial state and ending at the terminal state in the SUT, its state sequence is an executable path of the SUT. Also, we can use input conditions in the transition to design test cases and obtain the expected result of the SUT according to output results in the transition.

Definition 5 (Choice Operator): The choice operator is denoted by the notation "|", which indicates the choice relationship among some transition sequences.

The choice operator has the following seven properties [8]:

1) $\forall a,b \in \Sigma \bullet a|b \Rightarrow a \vee b$.
2) $\forall a,b \in \Sigma \bullet a|b = b|a$
3) $\forall a,b,c \in \Sigma \bullet a|b|c = (a|b)|c = a|(b|c)$
4) $\forall a \in \Sigma \bullet a|\varepsilon = \varepsilon|a = a$
5) $\forall a \in \Sigma \bullet a|a = a$
6) $\forall a,b_1,b_2,\ldots,b_n \in \Sigma \bullet a.(b_1|b_2|\ldots|b_n) = a.b_1|\, a.b_2|\ldots|\, a.b_n$
7) $\forall a_1,a_2,\ldots,a_n,b \in \Sigma \bullet (a_1|a_2|\ldots|a_n).b = a_1.b|\, a_2.b|\ldots|\, a_n.b$

By definition 5, we can get a set of transition sequences from the choice operation of transition sequences. For example, we can get transition sequences $a.b_1$, $a.b_2$, … , and $a.b_n$ from $a.(b_1|b_2|\ldots|b_n)$.

Definition 6 (Star Operator): The star operator is denoted by the notation "*", which indicates that the cycle number is from 0 to infinity.

By definition 6, if the model of the system can be described as a^*, it indicates that this SUT has infinite executable paths.

Definition 7 (Positive Operator): The positive operator is denoted by the notation "+", which indicates that the cycle number is from 1 to infinity.

By definition 7, the model a^+ of the SUT indicates that the system has to be executed at least once. Thus, there is an equation $a^+=a.a^*=a^*.a$. Although both the star operator and the positive operator have the ability

to describe two types of cycles in the SUT, they do not accurately describe all types of cycles. For instance, the statement "for(; i<3; i++)" indicates that a cycle may be 0, 1 or 2 times. To describe this type of cycle in the SUT, we give the definition of the range operator.

Definition 8 (Range Operator): The range operator is denoted by the notation {i~j}, which indicates that the number of cycles is from i to j, where both i and j satisfy 0≤i≤j.

By definitions 6, 7, and 8, both the star operation and the positive operation are two special types of the range operations. In fact, the regular expression a* is equal to $a^{\{0\sim n\}}$ if n is an infinite integer.

Definition 9 (Extended Regular Expressions): Extended regular expressions (ERE) make up of some notation in Σ and operators, including ".", "|", "*", "+", and {i≤j}.

3. Modeling Rules

To build the ERE model of the program, we designed six modeling rules to respectively deal with sequential statements, judgment statements, and loop statements of the program. To simplify the design of transitions, we use the number of lines of the program as both pre-states and next-states of transitions.

3.1 Modeling for Sequential statements

In this paper, sequential statements refer to assignment statements, variable assignment statements, and signal assignment statements.

Modeling rule 1 (Transition Rule): The sequential statement in the pre-state is taken as the input condition of the transition, and the variable on the left side of the equation in this sequential statement can be denoted as the output result of the transition.

Modeling rule 2 (Concatenation Rule): A set of adjacent sequential statements can be denoted by a transition sequence.

```
1  int x=getX();
2  int y=getY();
3  x=x^y;
4  y=y^x;
5  x=x^y;
6  printf(x);
```

Fig.1. Six sequential statements.

Fig. 1 shows a program segment with six sequential statements. According to modeling rules 1 and 2, we can obtain five transitions a=(1,x=getX()/x,2), b=(2,y=getY()/y,3), c=(3,x=x^y/x,4), d=(4,y=y^x/y,5), and e=(5,x=x^y/x,6). And the program segment in Figure 1 (a) can be modeled as $a.b.c.e.d$.

Modeling rule 3 (Block Rule): The transition sequence in a program block can be modeled by a new transition tr, where

- the pre-state of the transition tr is the pre-state of the first transition in the sequence,
- the next-state of the transition tr is the next-state of the last transition in the sequence,
- the input condition of the transition tr is the union set of all input conditions in the sequence, and
- the output result of the transition tr is the union set of all output results in the sequence.

Let k be $a.b.c.e.d$. Then, by modeling rule 3, there is k=(1, x=getX() ∧ y=getY() ∧ x=x^y ∧ y=y^x ∧ x=x^y/x ∧ y, 6).

3.2 Modeling for judgment statements

In this paper, judgment statements refer to both "if" and "switch" statements of the program. Then, we need to design modeling rules for two types of judgment statements.

Modeling rule 4 (Choice Rule): The judgment statement can be modeled by the choice operator.

Modeling rule 5 (Condition Rule): The decision condition of the judgment statement is the input condition of the transition and the output result is empty.

```
1 if(x<=y)
2    z=y;
   else
3    z=x;
```

(a)

```
4 switch(i){
5 case 1:
6   printf(x);break;
7 case 2:
8   printf(y);break;
9 case 3:
10  printf(y);break;
   }
```

(b)

Fig.2. Two decision statements.

Fig. 2 (a) is a program segment with the *if* statement, and Fig. 2 (b) is a program segment with the *switch* statement. By modeling rules 4 and 5, we can obtain two transitions a=(1, x<=y/-, 2) and b=(1, x>y/-, 3) for the program segment in Fig. 2 (a), and six transitions c=(4, i/-, 5), d=(4, i≠1/-, 7), e=(4, i≠1∧i≠2/-, 9), f=(5, i=1/x, 6), g=(7, i=2/y, 8), and h=(9, i=3/y, 10) for the program segment in Fig. Finally, the model $a|b$ can denote the program segment in Fig. 2 (a), and the model $a.d|b.e|c.f$ can denote the program statement in Fig. 2 (b).

3.3 Modeling for Loop Statements

In this paper, the loop statements refer to *while* statements, *for* statements, and *do…while* statements. We need to design modeling rules for three types of loop statements.

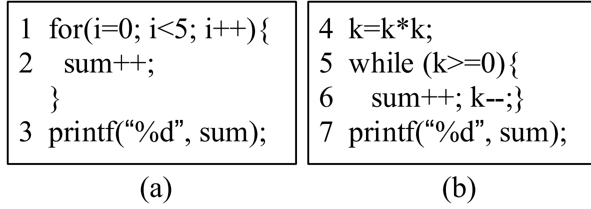


Fig.3. Two loop statements.

Modeling rule 6 (Loop Rule): If the variable(s) on the left of the decision conditions in the loop statement has (have) the infinite values, the star operator or the positive operator can describe this loop statement. If the decision conditions in the loop statement must be satisfied, the positive operator or the range operator can model the loop statement. If the finite values satisfy the decision conditions in the loop statement, the range operator can model this loop statement.

Fig.3 (a) is a program segment with the *for* statement, and Fig.3 (b) is a program segment with the *while* statement. By modeling rules 7, the model $(a.b)^{\{1\sim5\}}.c$ is constructed for the program segment with the *while* statement shown in Fig. 3 (a), and the model $a.(b.c)^{+}.d$ for the program segment with the *while* statement shown in Fig.3 (b).

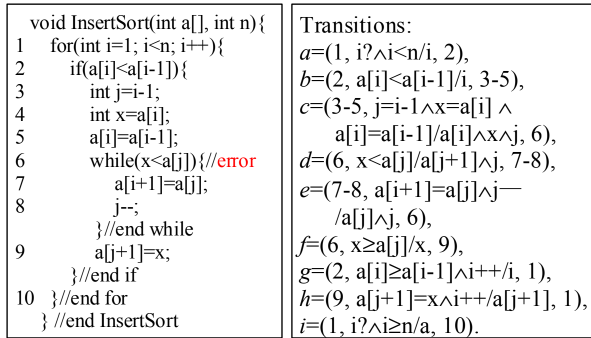


Fig.4. An insert sort algorithm and its transitions.

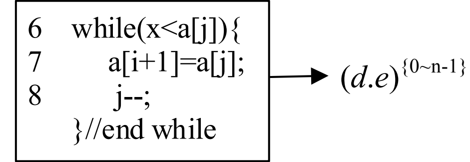
4. Case Study

In this section, we apply an insert sort algorithm to illustrate the processes of constructing the ERE model and test generation from this model. Fig.4 consists of an insert sort algorithm and a set of transitions, which are designed according to modeling rules 1 and 5. Also, there is an error in Line 6 of the insert sort algorithm. We will model this algorithm by the ERE model, and then test the algorithm by test cases generated from the ERE model so as to expose the error.

4.1 Modeling

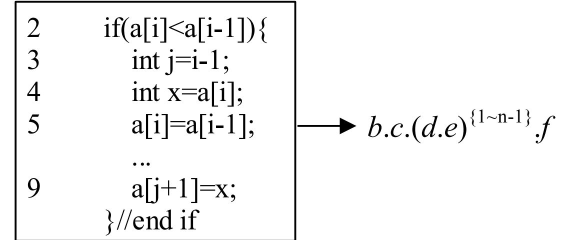
By six modeling rules in Section 3, we can convert the insert sort algorithm into the related ERE model. Our strategy is to convert the most internal statements in the algorithm first, and then the external statements are converted to the ERE model. So, we can apply three steps to model this algorithm as follows:

Step 1: Modeling for the *while* statement.



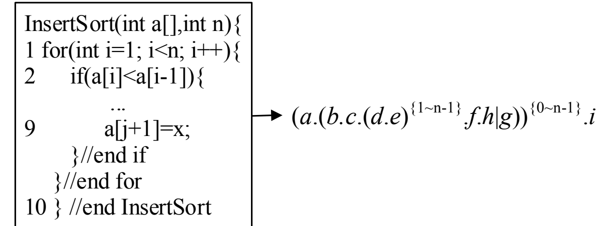
Since the cycle number in the *while* statement in Line 6 is from 0 to n-1 times, the program segment in **Step 1** is denoted by the model $(d.e)^{\{0\sim n\text{-}1\}}$.

Step 2: Modeling for the *while* statement.



In the program segment in **Step 2**, when the transition *b* occurs, the transition *d.e* must occur. Hence, the model $b.c.(d.e)^{\{1\sim n\text{-}1\}}.f$ can describe the program segment in **Step 2**.

Step 3: Modeling for the insert sort algorithm.



Since the cycle number of the *for* statement in Line 1 of the program segment in **Step 3** is from 0 to n-1, the algorithm can be denoted by the model $(a.(b.c.(d.e)^{\{1\sim n\text{-}1\}}.f.h \mid g))^{\{0\sim n\text{-}1\}}.i$.

4.2 Executable Path

To generate executable paths of the algorithm shown in Fig. 4, we need to deal with the ERE model $(a.(b.c.(d.e)^{\{1\sim n\text{-}1\}}.f.h \mid g))^{\{0\sim n\text{-}1\}}.i$ by using operation properties described in Section 2. Then, we can obtain $(a.(b.c.(d.e)^{\{1\sim n\text{-}1\}}.f.h \mid g))^{\{0\sim n\text{-}1\}}.i = (a.b.c.(d.e)^{\{1\sim n\text{-}1\}}.f.h \mid a.g)^{\{0\sim n\text{-}1\}}.i$.

Let A be $(a.b.c.(d.e)^{\{1\sim n-1\}}.f.h \mid a.g)$. Then, the ERE model of the insert algorithm is transformed to $A^{\{0\sim n-1\}}.i$. According to definition 8, we can obtain:

$A^{\{0\sim n-1\}}.i$

$= i \mid A.i \mid A^2.i \mid \ldots \mid A^{n-1}.i$

$= i \mid a.b.c.(d.e)^{\{1\sim n-1\}}.f.h.i \mid a.g.i \mid (a.b.c.(d.e)^{\{1\sim n-1\}}.f.h \mid a.g)^2.i \mid \ldots \mid (a.b.c.(d.e)^{\{1\sim n-1\}}.f.h \mid a.g)^{n-1}.i.$

Let B be $(d.e)^{\{1\sim n-1\}}$. Then, we can obtain:

$(a.b.c.(d.e)^{\{1\sim n-1\}}.f.h \mid a.g)^{\{0\sim n-1\}}.i$

$= i \mid a.b.c.(d.e)^{\{1\sim n-1\}}.f.h.i \mid a.g.i \mid (a.b.c.(d.e)^{\{1\sim n-1\}}.f.h \mid a.g)^2.i \mid \ldots \mid (a.b.c.(d.e)^{\{1\sim n-1\}}.f.h \mid a.g)^{n-1}.i$

$= i \mid a.g.i \mid a.b.c.B.f.h.i \mid (a.b.c.B.f.h \mid a.g)^2.i \mid \ldots \mid (a.b.c.B.f.h \mid a.g)^{n-1}.i$

In model-based testing, some coverage criteria [11] are used to generate test paths from the model. In this section, we select part of transition sequences in terms of the coverage criterion of independent paths [12]. Then, four transition sequences *i*, *a.g.i*, *a.b.c.B.f.h.i*, and *a.b.c.B.f.h.a.g.i* are obtained.

According to definition 8, there is $B=(d.e)^{\{1\sim n-1\}}=d.e \mid (d.e)^2 \mid \ldots \mid (d.e)^{n-1}$. Then, we can obtain:

$a.b.c.B.f.h.i$

$= a.b.c.(d.e \mid (d.e)^2 \mid \ldots \mid (d.e)^{n-1}).f.h.i$

$= a.b.c.d.e.f.h.i \mid a.b.c.d.e.d.e.f.h.i \mid \ldots \mid a.b.c.(d.e)^{n-1}.f.h.i$

By the coverage criterion of independent paths, we can get the transition sequence *a.b.c.d.e.f.h.i* from *a.b.c.B.f.h.i*. Similarly, we can obtain the transition sequence *a.b.c.d.e.f.h.a.g.i* from *a.b.c.B.f.h.a.g.i* to satisfy the independent path coverage criterion. Finally, four transition sequences *i*, *a.g.i*, *a.b.c.d.e.f.h.i*, and *a.b.c.d.e.f.h.a.g.i* are gained from the model $(a.(b.c.(d.e)^{\{1\sim n-1\}}.f.h \mid g))^{\{0\sim n-1\}}.i$.

For each transition sequence, we can construct state sequences as executable paths by using those states in transitions. Thus, four test executable paths are obtained from four transition sequences. Table I shows four transition sequences and four executable paths.

TABLE I. TRANSITION SEQUENCES AND THEIR EXECUTABLE PATHS FOR THE INSERT SORT ALGORITHM.

Transition Sequence	Executable Path
i	1-10
a.g.i	1-2-1-10
a.b.c.d.e.f.h.i	1-2-3-5-6-7-8-6-9-1-10
a.b.c.d.e.f.h.a.g.i	1-2-3-5-6-7-8-6-9-1-2-1-10

TABLE II. TEST SEQUENCES AND LISTS OF TEST INPUT CONDITIONS.

Transition Sequence	Input condition list
i	<i?∧i≥n>
a.g.i	<i?∧i<n, a[i]≥a[i-1]∧i++, i?∧ i≥n>
a.b.c.d.e.f.h.i	<i?∧i<n, a[i]<a[i-1], i, x<a[j], j--∧a[j], x≥a[j], x∧i++, i?∧i≥n>
a.b.c.d.e.f.h.a.g.i	<i?∧i<n, a[i]<a[i-1], i, x<a[j], j∧a[j], x≥a[j], x, i?∧i<n, a[i]≥a[i-1]∧i++, i?∧i≥n>

4.3 Instantiation

To design test cases of the insert sort algorithm, we need to apply some techniques of instantiation to generate test cases so as to cover those executable paths in Table 1. First, we construct a list of the input conditions for each transition sequence. Table 2 shows test sequences and the corresponding lists of test input conditions. Then, in terms of the formal reasoning method, we can simplify the test input condition sequence in Table 2 so as to design test cases.

For the list <i?∧i≥n>, there is $i?\wedge i\geq n \Rightarrow i=1\bullet 1\geq n$. Hence, when the size of the array a[n] is not greater than 1, the test case (a[n], n) will cover the executable path 1-10.

Theorem 1: we can obtain a[1]≥a[0]∧n=2 according to the list <i?∧i<n, a[i]≥a[i-1]∧i++, i?∧ i≥n>.

By theorem 1, when the size of the array a[n] is 2 and a[1] ≥a[0], the test case (a[n], n) covers the executable path 1-2-1-10.

Theorem 2: we can obtain a[1]<a[0]∧n=2 according to the list <i?∧i<n, a[i]<a[i-1], i, x<a[j], j--∧a[j], x≥a[j], x∧i++, i?∧i≥n>.

By theorem 2, when the size of the array a[n] is 2 and a[1]<a[0], the test case (a[n], n) covered the executable path 1-2-3-5-6-7-8-6-9-1-10.

Theorem 3: we can obtain a[1]<a[0] ∧ a[2]≥a[1] ∧ n=3 according to the list <i?∧i<n, a[i]<a[i-1], i, x<a[j], j--∧a[j], x≥a[j], x∧i++, i?∧i≥n>.

By theorem 3, when the size of the array a[n] is 3, a[1]<a[0], and a[2]≥a[1], the test case (a[n], n) covers the executable path 1-2-3-5-6-7-8-6-9-1-2-1-10. Finally, we design four test cases, shown in Table 3, for four executable paths according to theorems 1-3.

TABLE III. EXECUTABLE PATHS AND CORRESPONDING TEST CASES.

Executable Path	Test Cases
1-10	(a[0]=1, n=1)
1-2-1-10	(a[0]=3, a[1]=4, n=2)
1-2-3-5-6-7-8-6-9-1-10	(a[0]=4, a[1]=3, n=2)
1-2-3-5-6-7-8-6-9-1-2-1-10	(a[0]=4, a[1]=3, a[1]=5, n=3)

Through verification, test cases (a[0]=4, a[1]=3, n=2) and (a[0]=4, a[1]=3, a[1]=5, n=3) can detect the program error in Line 6 of the insert sort algorithm in Fig. 4.

4.4 Discussions

Generally, the model in model-based testing is stored as the graph in the computer. Then, test paths are constructed by means of graphical traversal algorithms. To compare the difference between the ERE-based testing and model-based testing, we use the method of [13] to construct a test tree of the model. In Fig. 5, there is a control flow graph of the insert sort algorithm shown in Fig. 4 and a test tree by traversing this graph. Then, four test paths 1-2-1, 1-2-3-5-6-7-8-6, 1-2-3-5-6-9-1, and 1-10 are obtained from the root node to the leaf nodes in the test tree. However, we find that test paths 1-2-1, 1-2-3-5-6-7-8-6, and 1-2-3-5-6-9-1 are incomplete because the leaf nodes 1 and 6 in the test tree are not the terminal node 10 of the insert sort algorithm. In addition, it is impossible to design a test case to cover the path segment 1-2-3-5-6-9-1. Therefore, traditional graph traversal algorithms have their limitations to construct executable paths from the graphical model.

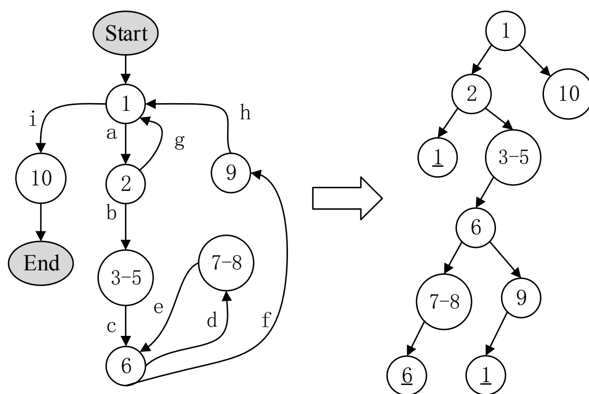


Fig.5. A control flow graph and its test tree.

5. Conclusions

The paper introduces an improved model-based test method, named as ERE-based testing, to generate test cases. The proposed method uses extended regular expressions to model the SUT. To construct the ERE model of a program, we have presented six modeling rules. Then, a case study is illustrated to how to apply these rules to set up the ERE model of the program and how to design test cases from the ERE model. In this study, four test cases are designed to satisfy the independent path coverage criterion, and two of designed test cases can detect the error in the insert sort algorithm. Compared with the traditional methods of generating test paths by traversing the model, ERE-based testing can construct test paths by using some algebraic operations. Moreover, executable paths constructed by our method are valid and complete and can be used to design effective test cases to detect program errors. In the future, we will perfect our previous work in the theory of the algebra system [8] and in the development of the test modeling tool [14] for constructing the ERE model.

References

[1] E. Amoroso, T. Nguyen, J. Weiss, J. Watson, P. Lapiska, and T. Starr, "Toward an approach to measuring software trust," in Research in Security and Privacy, 1991. Proceedings., 1991 IEEE Computer Society Symposium on, 1991, pp. 198-218.

[2] T. Grandison and M. Sloman, "A survey of trust in internet applications," *IEEE Communications Surveys & Tutorials,* vol. 3, pp. 2-16, 2000.

[3] J.-H. Cho, A. Swami, and R. Chen, "A survey on trust management for mobile ad hoc networks," *IEEE Communications Surveys & Tutorials,* vol. 13, pp. 562-583, 2011.

[4] A. Bertolino, "Software testing research: Achievements, challenges, dreams," in *2007 Future of Software Engineering*, 2007, pp. 85-103.

[5] R. V. Binder, B. Legeard, and A. Kramer, "Model-based testing: where does it stand?," *Communications of the ACM,* vol. 58, 2015.

[6] H. Hemmati, A. Arcuri, and L. Briand, "Achieving scalable model-based testing through test case diversity," *ACM Transactions on Software Engineering and Methodology (TOSEM),* vol. 22, p. 6, 2013.

[7] C. Schulze, D. Ganesan, M. Lindvall, R. Cleaveland, and D. Goldman, "Assessing model-based testing: an empirical study conducted in industry," in *Companion Proceedings of the 36th International Conference on Software Engineering*, 2014, pp. 135-144.

[8] P. Liu and H. Miao, "Theory of Test Modeling Based on Regular Expressions," in *Structured Object-Oriented Formal Language and Method*, ed: Springer, 2014, pp. 17-31.

[9] D. Kozen, "Kleene algebra with tests," *ACM Transactions on Programming Languages and Systems (TOPLAS),* vol. 19, pp. 427-443, 1997.

[10] J. Carlson and B. Lisper, "An event detection algebra for reactive systems," in *Proceedings of the 4th ACM international conference on Embedded software*, 2004, pp. 147-154.

[11] P. Ammann and J. Offutt, *Introduction to software testing*: Cambridge University Press, 2008.

[12] R. S. Pressman, *Software engineering: a practitioner's approach*: Palgrave Macmillan, 2005.

[13] H. Miao, Z. Qian, and B. Song, "Towards automatically generating test paths for Web application Testing," in *Theoretical Aspects of Software Engineering, 2008. TASE'08. 2nd IFIP/IEEE International Symposium on*, 2008, pp. 211-218.

[14] M. Zeng, P. Liu, and H. Miao, "The Design and Implementation of a Modeling Tool for Regular Expressions," in *Advanced Applied Informatics (IIAIAAI), 2014 IIAI 3rd International Conference on*, 2014, pp. 726-731.