Acta Technica *62* No. 1B/2017, 313–324

Software test data generation algorithm based on multi-dimensional space-time granularity

Wu Dafei[1]

Abstract. Based on the multi-dimensional space-time granularity, a kind of software test data generation algorithm was proposed. The algorithm first uses the space-time function multi-dimensional approach and granularity method, finds all feasible paths in the program. And it generates proper initial software test data set for the partial feasible paths automatically; when the correct software testing data can not be obtained by using the space-time function multi-dimensional approach and granularity method, it can depend on the principal of making software testing data set smallest and multi-dimensional space-time granularity thought. The software testing data can be supplemented according to the predicate and sub path that never covered by initial test data set. The new algorithm has a combination with predicate slice and DUC expression, so it is able to judge the feasibility of the sub path from the source. Then it can effectively to reduce the influence of unfeasible path on the algorithm performance. The algorithm analysis and experimental results show that the algorithm can effectively reduce the software test data bulk, and improve the test performance.

Key words. Software test, software test data automatic generation, multi-dimensional space-time granularity, multi-dimensional approach; space-time function granularity.

1. Introduction

At present, dynamic software test has become a research focus of software testing. In the dynamic software testing process, the generation of test cases is the key and difficulty of the task. According to statistics, about 40% of the test costs was spent on the design test cases [1]

Relying on a variety of test models and standards, the generation methods of software test data like dependent on the syntax, predicate slices, program specifications, symbol execution and program execution were proposed [2–6]. Path coverage is a typical test standard. The goal is to require that all paths of the program be tested at least once at the end of the test. In literature [7, 8], a software test data

[1]Hunan University of Science and Engineering, Yongzhou, Hunan, 425199, China

generation algorithm based on path coverage test is given by symbolic execution, as the implementation of the symbol on the form of the array and the pointer is difficult to deal with; In literature [9, 10], program execution and space-time function granularity are used to solve the drawbacks of the array execution and pointers. However, since each test only considers one predicate and one input variable, a large number of iterations are required to automatically generate a new input that satisfies the condition even if the branch condition in the path is a linear function of the input. A new algorithm for a given path-dependent iteration is proposed in Literature [11]. The algorithm takes into account multiple predicates and multiple input variables at same time. A set of initial inputs satisfying the given path is automatically generated by iteration using a randomly selected set of initial inputs. The algorithm only uses the linear function to iterate, so it is only effective for the linear function, and less effective for the nonlinear function.

In this paper, a software test data generation algorithm based on the idea of multi-dimensional space-time granularity was proposed. The new algorithm first uses the multidimensional approximation and granularity method of the space-time function to find out the feasible path of the program and automatically generate the appropriate initial software test data set for some feasible paths. When the space-time function multi-dimensional approach and granularity methods can not get the software test data correctly, it can depend on the principal of making software testing data set smallest and multi-dimensional space-time granularity thought for the predicate and sub path that never covered by initial test data set. The software testing data can be supplemented according to the thought. Since the new algorithm uses the DUC expression [6], it is possible to determine whether the sub path is feasible from the source, so as to effectively reduce the influence of unfeasible paths on the algorithm. The new algorithm combines the advantages of space-time function multi-dimensional approach and granularity methods. And it relies on multi-dimensional space-time granularity to supplement the software test data, thus reducing the number of software test data and improving the test performance.

Definition 1. Path. A program module M can be regarded as a directed graph $G = (V, E, s, e)$, where V is the set of nodes in M, E is the set of edges in M, s is the unique source node of M, indicating the beginning of the program module M; e represents the only sink nodes for M, meaning the end of program module M. A node n means a piece of declaration statements or a conditional expression. From node n_i to node n_j, a possible control transfer is regard as an edge $(n_i, n_j) \in G$. A subpath $P = \{n_1, n_2, \cdots, n_{k+1}\} \in G$ is a sequence of node n_i, and $(n_i, n_{i+1}) \in E$, $\forall \in [1, k]$; if $n_1 = s, n_{k+1} = e$, then P is called a piece of path in program module M.

Definition 2. Input variable. If a variable i_k appears in an input statement program of program module M or an input parameter of M, then i_k is called the input variable of i_k. The definition domain D_k of input variable i_k is a set for all the possible values of i_k. An input vector $I = \{i_1, i_2, \cdots, i_m\} \in (D_1 \times D_2 \times \cdots \times D_m)$ is called a program input of program module M, which is called an input of M for short. Herein, m is the number of the input variables in M.

Definition 3. Predicate. Simple predicate refers to the predicate only containing

single relational operator *op*; obviously, the simple predicate can be expressed as $E_1 op E_2$. Herein, $op \in \{<, \leq, >, \geq, =, \neq\}, E_1, E_2$ is the algebraic expression. Compound predicate means the predicate of two or more than two simple predicates connected by Boolean connectives AND or OR together. If there is a boolean variable in the predicate, then TRUE can be expressed by 0 or a positive number, and FALSE can be expressed by a negative number.

Definition 4. An unfeasible path. All predicate explanations [8] on a subpath compose its path condition, which defines a path definition field. It ensures the program to consist of all inputs that execute along the subpath. If the path definition field of a sub-path is empty (ϕ), the sub-path is called an unfeasible sub-path.

The following *Lemma 1* is clearly defined by the *Definition 4*: from *Definition 4*, it is clear that the following *Lemma 1* comes into existence.

Lemma 1. If a sub-path is an unfeasible sub-path, then all paths containing the sub-path are unfeasible paths.

Definition 5. Space-time function. A simple predicate $pr : E_1 op E_2$ can be converted into the following form: $F\ rel\ 0$; herein: $rel \in \{<, \leq, =\}; F$ is a direct or indirect function of the input variable, and called the corresponding space-time function of the predicate pr.

Definition 6. Multi-dimensional approach. For arbitrary space-time function F_1 and a set of inputs I_0, it is called the multi-dimensional approach of linear function $f_1 = \sum_{i=1}^{m} a_i^1 x_i + b_1$ for F_1. Herein, x_i is the input variable; a_i^1 is the constant coefficient; b_1 is the constant; a_i^1 and b_1 can be obtained from the following equations:

$$\begin{cases} f_1(I_0) = F_1(I_0), \\ f_1(I_0 + (\Delta_{x_1}, 0, \cdots, 0)) = \\ F_1(I_0 + (\Delta_{x_1}, 0, \cdots, 0)), \\ \vdots \\ f_1\left(I_0 + \left(0, \cdots, 0, \Delta_{x_{m-1}}, 0\right)\right) = \\ F_1\left(I_0 + \left(0, \cdots, 0, \Delta_{x_{m-1}}, 0\right)\right), \\ f_1(I_0 + (0, \cdots, 0, \Delta_{x_m})) = \\ F_1(I_0 + (0, \cdots, 0, \Delta_{x_m})). \end{cases}$$

Theorem 1. For a given input I_0, supposing the corresponding multi-dimensional approach f_1 of the space-time function F_1 for the predicate p_1 is f_1, and the operator is of F_1 is rel_1, then: 1) if $f_1(I_0)\, rel_1 0 \Rightarrow I_0$ ensures the program to execute along the sub-path p_1; 2) if $f_1(I_0)\, rel_1 0 \Rightarrow I_0$ ensures the program to execute along the sub path p_1.

Proof. It can be seen from Definition 6: $f_1(I_0) = F_1(I_0)$, then $f_1(I_0)\, rel_1 0$ is set

up $\Rightarrow F_1(I_0)\, rel_1 0$ is set up $\Rightarrow I_0$ ensures the program to execute along the sub path p_1. Proving by the same method, $f_1(I_0)\, \neg rel_1 0$ is set up $\Rightarrow I_0$ ensures the program to execute along the sub-path p_1, Q.E.D.

2. Software test data generation algorithm based on multi-dimensional space-time granularity

2.1. Generation algorithm of initial software test data set

Based on the definitions, theorems and lemmas in section 1, the initial software test data set generation algorithm is described as follows:

Step 1. By using the predicate slicing algorithm [6], the DUC expression $r_j.d_j : U_j : c_j$ and operator rel_j corresponding in M of all the n predicates $p_j\,(j \in [1,n])$ were obtained; and all the n dicerse input variables $x_1, x_2, \cdots, x_m$ in M are obtained.

Step 2. Randomly generate a program input $I_0 = \left(x_1^0, x_2^0, \cdots, x_m^0\right)$, order $I^0 = I_0$.

Step 3. For $p^k\,(k \in [1,n])$ and input $I^{t_1 t_2 \cdots t_{k-1}}$ (herein, $t_i \in \{0,1\}, i \in [1,k-1]$, and when $k=1$, order $I^{t_1 t_2 \cdots t_{k-1}} = I^0$.

Step 3.1. If $k \geq 2$ and $\sim p_1^{t_1}, \sim p_2^{t_1 t_2}, \cdots, \sim p_{k-1}^{t_1 t_2 \cdots t_{k-1}} \notin c_k$, then skip to Step 3.2, otherwise skip to Step 3.5.

Step 3.2. By using *Definition 6*, according to the input $I^{t_1 t_2 \cdots t_{k-1}}$, the multi-dimensional approach f_k of p^k corresponded space-time function F_k can be solved. Substitute $I^{t_1 t_2 \cdots t_{k-1}}$ into $f_1^{t_1}, f_2^{t_1 t_2}, \cdots, f_{k-1}^{t_1 t_2 \cdots t_{k-1}}, f_k$ (herein $f_i^{t_1 t_2 \cdots t_{i-1}} = f_i, i \in [1,k-1]$) and obtain a group of values $\theta = (\theta_1, \theta_2, \cdots, \theta_k)$.

Step 3.3. If $\theta_k\ rel_k 0$, then $I^{t_1 t_2 \cdots t_{k-1}}$ ensures the program execute along the sub-path $p_1^{t_1}, p_2^{t_1 t_2}, \cdots, p_{k-1}^{t_1 t_2 \cdots t_{k-1}}, p_k$. Order $I^{t_1 t_2 \cdots t_{k-1}^0} = I^{t_1 t_2 \cdots t_{k-1}}, f_k^{t_1 t_2 \cdots t_{k-1}^0} = f_k, p_k^{t_1 t_2 \cdots t_{k-1}^0} = p_k$, and record the path and its input as well as the multi-dimensional approach set of space-time function; then, solve the vector I relying on *Theorem 2*, order $I_1 = I^{t_1 t_2 \cdots t_{k-1}} + I$.

Step 3.3.1. If I_1 ensures the program execute along the sub-path $p_1^{t_1}, p_2^{t_1 t_2}, \cdots, p_{k-1}^{t_1 t_2 \cdots t_{k-1}}, \sim p_k$, then order $I^{t_1 t_2 \cdots t_{k-1}^1} = I_1, f_k^{t_1 t_2 \cdots t_{k-1}^1} = f_k, p_k^{t_1 t_2 \cdots t_{k-1}^0} = \sim p_k$, and record the path and its input as well as the multi-dimensional approach set of space-time function; and then skip to Step 3.5.

Step 3.3.2. Otherwise, the solution of the program executing along the above subpath can be found according to the space-time function granularity thought. If it is able to find the input $I_1^{'}$ meeting the conditions in a given maximum number (if cannot, then order $I_1^{'} = \phi$), then use the *Definition 6* and obtain the new multi-dimensional approach $f_1^{'}, f_2^{'}, \cdots, f_{k-1}^{'}, f_k^{'}$ of $F_1, F_2, \cdots, F_k$, order $I^{t_1 t_2 \cdots t_{k-1}^1} = I_1^{'}, f_1^{t_1} = f_1^{'}, f_2^{t_1 t_2} = f_2^{'}, f_{k-1}^{t_1 t_2 \cdots t_{k-1}} = f_{k-1}^{'}, f_k^{t_1 t_2 \cdots t_{k-1}^1} = f_k^{'}, p_k^{t_1 t_2 \cdots t_{k-1}^1} = \sim p_k$, and record the path and its input as well as the multi-dimensional approach set of space-

time function; and then skip to Step 3.5.

Step 3.4. If θ_k $rel_k 0$ is not set up, then $I^{t_1t_2\cdots t_{k-1}}$ ensures the program execute along the sub-path $p_1^{t_1}, p_2^{t_1t_2}, \cdots, p_{k-1}^{t_1t_2\cdots t_{k-1}}, \sim p_k$ order $I^{t_1t_2\cdots t_{k-1}^1} = I^{t_1t_2\cdots t_{k-1}}$, $f_k^{t_1t_2\cdots t_{k-1}} = f_{k-1}^{'}, f_k^{t_1t_2\cdots t_{k-1}^1} = f_k^{'}, p_k^{t_1t_2\cdots t_{k-1}^1} = \sim p_k$, and record the path and its input as well as the multi-dimensional approach set of space-time function; then, solve the solution vector I according to Theorem 2, order the input $I_1 = I^{t_1t_2\cdots t_{k-1}} + I$. Finally, record the path and its input as well as the multi-dimensional approach set of space-time function; and then skip to Step 3.5.

Step 3.5. For another set of $t_1t_2\cdots t_{k-1}$, repeat the Step 3.1 to 3.4, until all the groups in $t_1t_2\cdots t_{k-1}$ are tested, and then skip to Step 4.

Step 4. Repeat the Step 3 for $k = k + 1$, till $k = n$, and then skip to Step 5.

Step 5. End of the algorithm: for record of the obtained path and its input as well as the multi-dimensional approach set of space-time function, if the path and input are not empty, then the path is a feasible path, and the corresponding input is a software test data meeting the path.

2.2. Supplement algorithm of software test data

Considering the test overhead, it is impractical to construct the appropriate software test data for all paths in the program module M. Therefore, in this section, we propose a new software test data addition algorithm, which is based on the idea of multi-dimension. Before the description of a specific algorithm is given, the relevant definitions are given as follows:

Definition 7. Sub-path pair. For software test data t_i, assume that the corresponding test path is $P_i = b_{i1}b_{i2}\cdots b_{it}$, herein,

$$b_{i1}b_{i2}\cdots b_{it} \in \{p_1, \sim p_1, p_2, \sim p_2, \cdots, p_k, \sim p_k\}, p_1, p_2, \cdots, p_k$$

is all k predicates in program module M, then call (b_{i1}, b_{i2}) is the subpath pair covered by software test data t_i, herein $j \in [1, t-1]$. The subpath pair set covered by software test data t_i is recorded as $\Psi_i = b_{i1}, b_{i2}\{(b_{i1}, b_{i2}), (b_{i2}, b_{i3}), \cdots, (b_{it-1}, b_{it})\}$. According to the software test data generation algorithm in Section 2.1, an initial software test data set can be obtained. Multi-dimensional space-time granularity refers to that based on the initial software test data set, first supplement the software test data for uncovered predicate. And then selectively supplement the redundancy software test data relying on the idea of subpath pair covering, in order to achieve better test coverage.

According to the above definition, the supplementary algorithm can be described as follows:

1) Assume that after using the initial software test data set generation algorithm, the obtained initial software test data set is $\{t_1, t_2, \cdots, t_n\}$, and the covered paths are $\{P_1, P_2, \cdots, P_N\}, p_1, p_2, \cdots, p_k$ are all the k predicates in program module M. Assume that the set of P_j covered predicates is Ω_j. If i exists and makes $p_i \notin \bigcup_{j=1}^{N} \Omega_j$

or $\sim p_i \notin \bigcup_{j=1}^{N} \Omega_j \mathrm{j}$, then supplement new software test data $t^{'}$ for p_i or $\sim p_i$.

2) Assume the obtained software test data set is $\{t_1, t_2, \cdots, t_m\}$ after the supplement of software test data in Step 1. For any software test data $t_i \in \{t_1, t_2, \cdots, t_m\}$, assume the covered sub-path pair set is Ψ_i. Then the sub-path pair set covered by software test data set $\{t_1, t_2, \cdots, t_m\}$ is $\Psi = \bigcup_{j=1}^{M} \Psi_i$. If sub-path pair set $\{(b_1, b_2), (b_2, b_3), \cdots, (b_{x-1}, b_x)\} \in \Psi$ exists, but $\forall \in [1, M]$, there is

$$\{(b_1, b_2), (b_2, b_3), \cdots, (b_{x-1}, b_x)\} \notin \Psi,$$

then supplement new software test data t" for sub-path $b_1, b_2, b_3 \cdots, b_x$.

3. Algorithm analysis and experiment

The complexity of the algorithm is mainly concentrated in the initial software test data set generation algorithm Step 3. In the worst case, for a program module containing n predicates, the worst-case algorithm has a time complexity of $O(2^n)$, since there may be at most 2^n different paths in the program. However, in practice, the actual complexity of the algorithm is much lower because the condition $\sim p_1^{t_1}, \sim p_2^{t_1 t_2}, \cdots, \sim p_{k-1}^{t_1 t_2 \cdots t_{k-1}} \notin c^k$ in Step 3.1 of the initial software test dataset generation algorithm will cause a large number of unfeasible paths to be removed in time. In addition, the method of automatically generating software test data by using the spatio-temporal function granularity method, even for the linear function, needs to be tested several times in order to find the appropriate software test data; The new algorithm proposed in this paper combines the advantages of both the multidimensional approximation and the spatio-temporal function granularity. Therefore, the new algorithm requires only one iteration of the linear function to get the appropriate software test data. The use of non-linear function, multidimensional approximation and spatio-temporal function granularity method makes the new algorithm converge faster from the initial input to the satisfying software test data. Moreover, the new algorithm relies on the idea of multidimensional spatio-temporal granularity to update the software test data for predicates and sub-paths that are not covered by the initial software test dataset, So there is better test coverage performance. The program module M_1 is illustrated as an example, and the program module M_1 is shown in Fig. 1.

For M_1, according to the algorithm described in Section 2, the specific implementation steps are as follows.

First solve the DUC expression of program module M_1. Assume the automatically generated initial input is $I_0 = (1, 2)$.

For p_1, it can be seen from algorithm Step 3.1 that I_0 ensures the execution of $\sim p_1$. According to algorithm Step 3.4, another group of solutions $I_1 = (2, 0)$ can be obtained. And I_1 ensures the execution of p_1. Then get $\Gamma \leftarrow < -x + y, p_1, (3, 2) >$.

For p_2: for the record 1 in Γ $1 :< -x + y, \sim p_1, (1, 2) >$, it can be seen from algorithm Step 3.1 that (1, 2) ensures the execution of $\sim p_2$. Record $\Gamma \leftarrow <$

$\{-x+y, x+3y\}, \{\sim p_1, \sim p_2\}, (1,2) >$, then another group of solutions $I_1 = (-1, 0)$ can be obtained from algorithm Step 3.2 and 3.3; obviously, (–1, 0) ensures the execution of path $\sim p_1, p_2$. So

$$\Gamma \leftarrow < \{-x+y, x+3y\}, \{\sim p_1, \sim p_2\}, (-1,0) > .$$

r	d	U	c
$read(x,y)$	x,y		
if $(x>y)$ then	p_1	x,y	
$w = x + 2^* y$	w	x,y	p_1
else			
$w = y$	w	y	: p_1
endif			
if $(w+y<0)$ then	p_2	w,y	
$x = x-2$	x	x	p_2
$y = y+w$	y	w,y	p_2
$write(y)$	o_1	y	p_2
else			
if $(x+y-y^3<0)$ then	p_3	w,y	: p_2
$write(y)$	o_2	y	p_3
else			
$write(w)$	o_3	w	: p_3
endif			
endif			

Fig. 1. Program module M_1

For the record 2 in $2 :< -x+y, p_1, (3,2) >$, we also can get

$$\Gamma \leftarrow < \{-x+y, x+3y\}, \{p_1, p_2\}, (0.5, -0.5) >$$

and $\Gamma \leftarrow < \{-x+y, x+3y\}, \{p_1, \sim p_2\}, (3,2) >$.

For p_3, since $c_3 = \{\sim p_2\}$, so it can be known from algorithm Step 3.4 that only two pieces of records need to be considered in Γ, $< \{-x+y, x+3y\}, \{\sim p_1, \sim p_2\}, (1,2) >$.

For $< \{-x+y, x+3y\}, \{\sim p_1, \sim p_2\}, (1,2) >$), we can only obtain

$$\Gamma \leftarrow < \{-x+y, x+3y, x-4y+6\}, \{\sim p_1, \sim p_2, p_3\}, (1,2) > ;$$

while for path $\{\sim p_1, \sim p_2, \sim p_3\}$, it cannot solve the software test data meeting the condition.

For $< \{-x+y, x+3y\}, \{p_1, \sim p_2\}, (3,2) >$, we can only obtain

$$\Gamma \leftarrow < \{-x+y, x+3y, x-4y+6\}, \{p_1, \sim p_2, p_3\}, (3,2) > .$$

However, for path $\{\sim p_1, \sim p_2, \sim p_3\}$, it cannot solve the software test data meeting the condition.

The obtained paths $\Gamma \leftarrow < \{-x+y, x+3y, x-4y+6\}, \{\sim p_1, \sim p_2, p_3\}, (1,2) >$ are feasible paths, and the corresponding software test data are t_1=(0.5, −0.5), t_2=(−1, 0), t_3=(3 , 2) and t_4=(1, 2), respectively. The covered sub-path pairs and predicates are shown in Tables 1 and 2.

Table 1. Subpaths covered by initial software test data set

Test data	(p_1, p_2)	(p_1, $\sim p_2$)	($\sim p_1$, p_2)	($\sim p_1$, p_2)	(p_2, p_3)	($\sim p_2$, p_3)
t_1	*					
t_2			*			*
t_3		*				*
t_4				*		*

Table 2. Predicates covered by initial software test data set

Test data	p_1	$\sim p_1$	p_2	$\sim p_2$	p_3	$\sim p_3$
t_1	*		*			
t_2		*	*			
t_3	*			*	*	
t_4						

Table 1 and Table 2 show that the sub-path $\sim p_3$ is not covered by t_1, t_2, t_3, t_4. According to Step 1 in software test data supplementary algorithm, the obtained software test data set is the subpath pairs and predicate covered by

$$\{t_1(0.5, -0.5), t_2(-1, 0), t_3(3, 2), t_4(1, 2), t_5(2, 1), t_6(1, 1)\},$$

which is shown in Tables 3 and 4.

Table 3. Sub-paths covered by software test data set after the first supplement

Test data	(p_1,p_2)	(p_1,$\sim p_2$)	($\sim p_1$,p_2)	($\sim p_1$,$\sim p_2$)	(p_2,p_3)	($\sim p_2$,p_3)	($\sim p_2$,$\sim p_3$)
t_1	*						
t_2			*			*	
t_3		*				*	
t_4				*		*	
t_5		*					*

Table 4. Predicates covered by software test data set after the first supplement

Test data	p_1	$\sim p_1$	p_2	$\sim p_2$	p_3	$\sim p_3$
t_1	*		*			
t_2		*	*			
t_3	*			*	*	
t_4						
t_5						

Table 3 and Table 4 show that the sub-paths $(\sim p_1, \sim p_2)$,$(\sim p_2, \sim p_3)$ meet the conditions of Step 2 in software test data supplementary algorithm. Therefore, software test data should be added. Assume that the new added software test data is $t_6(1,\ 1)$, then the obtained software test data set is the subpath pairs and predicate covered by $\{t_1\,(0.5, -0.5)\,, t_2\,(-1, 0)\,, t_3\,(3, 2)\,, t_4\,(1, 2)\,, t_5\,(2, 1)\,, t_6\,(1, 1)\}$, which is shown in Tables 5 and 6.

Table 5. Sub-paths Subpaths covered by software test data set after the second supplement

Test data	(p_1,p_2)	(p_1,$\sim p_2$)	($\sim p_1$,p_2)	($\sim p_1$,$\sim p_2$)	(p_2,p_3)	($\sim p_2$,p_3)	($\sim p_2$,$\sim p_3$)
t_1	*						
t_2			*			*	
t_3		*				*	
t_4				*		*	
t_5		*					*
t_6				*			*

Table 6. Predicates covered by software test data set after the second supplement

Test data	p_1	$\sim p_1$	p_2	$\sim p_2$	p_3	$\sim p_3$
t_1	*		*			
t_2		*	*			
t_3	*			*	*	
t_4						
t_5	*				*	*
t_6		*		*		*

Obviously, Table 5 and Table 6 no longer meet the conditions of the software test data supplement algorithm. Thus, the finally obtained software test data set is $\{t_1\,(0.5, -0.5)\,, t_2\,(-1, 0)\,, t_3\,(3, 2)\,, t_4\,(1, 2)\,, t_5\,(2, 1)\,, t_6\,(1, 1)\}$.

In particular, the software test data

$$\{t_1\,(0.5, -0.5)\,, t_2\,(-1, 0)\,, t_3\,(3, 2)\,, t_4\,(1, 2)\,, t_5\,(2, 1)\,, t_6\,(1, 1)\}$$

obtained in this case covers all the 6 pieces of feasible paths $p_1p_2, \sim p_1p_2, p_1 \sim p_2p_3, \sim p_1 \sim p_2p_3$, $p_1 \sim p_2 \sim p_3$ and $\sim p_1 \sim p_2 \sim p_3p_1 \sim p_2 \sim p_3$ in the program. Finally, in order to further verify the performance of the algorithm, experiments were conducted depending on the test pool data in literature [12–14] . The results are shown in Fig. 2. The test results are the average value of each group of program test.

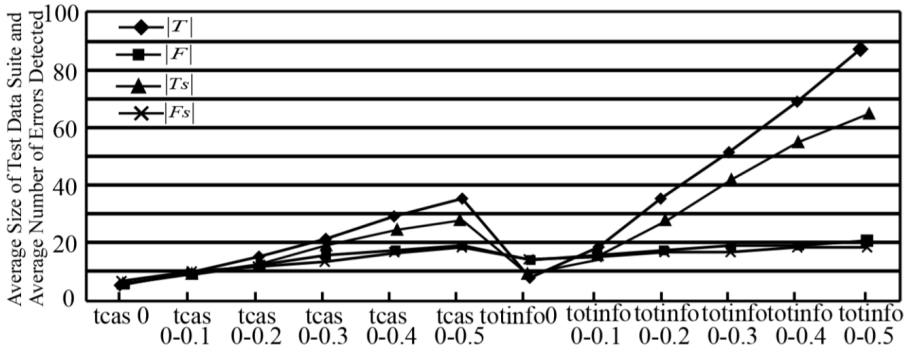

Fig. 2. Average size of the software test data set and the average number of detected errors

In Fig. 2, $|T|$ represents the average size of the original software testing data set. $|F|$ represents the average number of detected errors with the original software test data set. Quantities $|T_s|$ and $|F_s|$ mean the average number of average size and error detection software test data obtained by using the proposed algorithm, respectively. The experimental results show that the novel algorithm proposed in this paper can effectively reduce the number of software test data on the basis of guarantee the testing performance.

4. Conclusion

Software test is a very cumbersome and complex but extremely important stage in software development, especially for large-scale system software and application software. If the potential errors and defects in the software are not detected in time, they will cause serious consequences. In this paper, a software test data generation algorithm based on the idea of multi-dimensional space-time granularity was proposed. The algorithm analysis and experimental results show that the new algorithm can effectively reduce the influence of the unfeasible path to the algorithm, reduce the number of software test data, and improve the test efficiency. The next step is mainly to approach and solve approximation appropriate software test data for complex calculus functions, and to test software of structured program containing the string and numerical calculation and to generate test data automatically.

References

[1] H. Y. Chen, T. H. Tse: *Equality to equals and unequals: A revisit of the equivalence and nonequivalence criteria in class-level testing of object-oriented software.* IEEE Transactions on Software Engineering *39* (2013), No. 11, 1549–1563.

[2] J. W. Cangussu, R. A. DeCarlo, A. P. Mathur: *A formal model of the software test process.* IEEE Transactions on Software Engineering *28* (2002), No. 8, 782–796.

[3] T. J. Ostrand, E. J. Weyuker, R. M. Bell: *Predicting the location and number of faults in large software systems.* IEEE Transactions on Software Engineering *31* (2005), No. 4, 340–355.

[4] H. Li, S. Ding: *Research of individual neural network generation and ensemble algorithm based on quotient space granularity clustering.* Applied Mathematics & Information Sciences *7* (2013), No. 2, 701–708.

[5] X. Yao, D. Gong, W. Wang: *Test data generation for multiple paths based on local evolution.* Chinese Journal of Electronics *24* (2015), No. 1, 46–51.

[6] R. L. Zhao, Y. H. Min: *Automatic test data generation of character string based on predicate slice.* Journal of Computer Research and Development *39* (2002), No. 4, 473–481.

[7] C. Chiu, P. L. Hsu: *A constraint-based genetic algorithm approach for mining classification rules.* IEEE Transactions on Systems, Man, and Cybernetics, Part C: (Applications and Reviews) *35* (2005), No. 2, 205–220.

[8] A. Rataj, M. Nowak, P. Pecka: *Modelling CTMC with a standard programming language and using conventions from computer networking.* Theoretical and Applied Informatics *23* (2011), Nos. 3–4, 229–243.

[9] W. Miller, D. L. Spooner: *Automatic generation of floating-point test data.* IEEE Transactions on Software Engineering *SE–2* (1976), No. 3, 223–226.

[10] M. J. Gallagher, V. L. Narasimhan: *ADTEST: A test data generation suite for Ada software systems.* IEEE Transactions on Software Engineering *23*, (1997), No. 8, 473–484.

[11] N. Gupta, A. P. Mathur, L. M. Soffa: *Automated test data generation using an iterative relaxation method.* International Symposium on Foundations of Software Engineering (SIGSOFT), 1–5 November 1998, Lake Buena Vista, FL, USA, ACM SIGSOFT Software Engineering Notes *23* (1998), No. 6, 231–244.

[12] M. J. Balcer, W. M. Hasling, T. J. Ostrand: *Automatic generation of test scripts from formal test specifications.* Symposium on Software Testing, Analysis, and Verification (TAV3), 13–15 December 1989, Key West, FL, USA, ACM SIGSOFT Software Engineering Notes *14* (1989), No. 8, 210–218.

[13] D. Shin, E. Jee, D. H. Bae: *Comprehensive analysis of FBD test coverage criteria using mutants.* Software & Systems Modelings *15* (2016), No. 3, 631–645.

[14] D. Bideau, R. Hékinian: *A dynamic model for generating small-scale heterogeneities in ocean floor basalts.* Journal of Geophysical Research *100* (1995), No. B6, 10141 to 10162.

Received June 29, 2017