A Test Sequence Generation Method Based on Dependencies and Slices

Jin-peng MO*, Jun-yi LI and Jian-wen HUANG
College of Computer Science and Electronic Engineering, Hunan University,
Changsha 410082, China
*Corresponding author

Keywords: Test sequence, Dependence, Test case slice, Resetting of database.

Abstract. Based on the incorrect test results caused by dependencies in the execution of test cases, this paper proposes a method to optimize test sequence. Extract the dependencies to construct the dependence loop set, separate with iteration method in the maximum loop-cutting way to obtain a test case slice without dependence loops, and repeat the above step to get all the test case slices. For the test case slice set, get the node dependence value of each test case and sort the values in a descending order to generate the test sequence. The algorithm analysis and experimental results show that compared with the optimization techniques considering dependencies, this method reduces the number of resetting of database.

Introduction

The test sequence generation is an optimization technique includes designing the test cases in accordance with the importance and correlation of the function and the precondition of data and executing the test cases in a certain order. At present, the test sequence optimization technology mainly concentrates on three aspects, i.e. selecting the test cases, minimizing the test cases and sorting the prioritization of the test cases.

Rothermel et al. [1] proposed a test sequence optimization technique based on covering, and sorted the test cases according to their ability to detect faults. Briand [2] first analyzed the inter-class dependencies and circulation problems, devised the combinatorial test sequence generation method based on the coupling measurement and genetic algorithms. Do et al. [3] applied the test suite to the JUnit test framework under the Java language, significantly improved the fault-detection rate of JUnit test suite. Srikanth et al. [4][5] pointed out that the test suite prioritization depends on the code coverage information or historical data, and proposed test sequence generation by the relationship of fault-detection in order to make the examination process of the test case driven by algorithm. Mei et al. [6] sorted the test cases by analyzing the static call graph and the dynamic results and calculating the ratio of the code enforcement of each test case. Based on the protocol conformance problems caused by the dynamic state transition of the extended finite state machine model (EFSM), You et al. [7] proposed the test sequence generation algorithm in first search algorithm by edge weight. Zhao et al. [8] constructed an object-oriented complex network in the idea of inter-class integration, representing the classes with nodes and the inter-class relationship with edges, proposed a node measurement method by analyzing the influence and complexity of the nodes. As for how to sort the test cases, Chen and Gao [9] proposed a static test case sequence generation method based on the cyclomatic complexity. Zhang et al. [10] proposed a class integration testing order determination method based on particle swarm optimization (PSO), the algorithm reduces the overall complexity of structuring test piles in virtue of local optimality of PSO.

The previous studies of test sequence generation method mainly focused on minimizing the test cases or sorting the test cases by giving weight to the importance. Nevertheless, there few researches on the mutual influence between the test cases, i.e. the test case dependencies. In the software testing process, the test cases mostly can interact with the database and modify the status of database, resulting in the incorrect test results[11].

In the test process of database application system, Liang et al. [12] fully considered the influence of the status of database on the test cases, analyzed the dependencies with UML class diagrams,
separated the test case set in slicing, and then constructed the test sequence generation algorithm. Based on the research of Liang et al. [12], Chen [13] improved the algorithm by, under the condition of the least number of slices, ensuring that no loops exist in the directional dependence diagram constructed by the test cases in the slice, and by using the reverse topological sorting for each slice to generate the test sequence. While these methods preliminarily solve the test conflict caused by dependencies, the loop-breaking in the process of accessing the slices results in low efficiency.

Considering the above, this paper puts forward a test sequence generation method based on the dependencies and slices. This new method generates test sequence by extracting the dependencies of the test cases and by eliminating the dependencies by means of slicing.

Related Theory

Node Dependence Value

During the execution of the test cases, some executed test cases can affect the state of database, destroying the preconditions of other test cases, and resulting in incorrect test results. Based on the dependencies between the test cases, this paper adopts the node dependence value to quantify the executive priority of the test cases.

Definition 1 Dependence (D): Let $T_i$ and $T_k$ be any two elements in the test case set $TS = \{T_1, T_2, ..., T_n\}$[12], when the execution of $T_i$ has influence on $T_k$, there is a dependence between $T_i$ and $T_k$, expressed as

$$D = <T_i, T_k>.$$  

The dependence in the above expression is expressed as: $T_i$ depends on $T_k$, $T_i$ is called initial node and $T_k$ is called terminal node. All the dependencies in test case set constitute a dependence set (DS), expressed as

$$DS = \{D_i | i = 1, 2, ..., m\}. \tag{2}$$

where, $m$ represents the number of dependencies. Assuming that $T_{DS}$ is a subset of DS and the initial node and terminal node of all dependencies in $T_{DS}$ belong to a test case set, then $T_{DS}$ is called the dependence set corresponding to this test case set.

For dependence, this paper specifies that $<T_i, T_k>$ and $<T_k, T_i>$ do not exist at the same time, that is, there is only one dependence or no dependence between two test cases. Therefore, the test case set containing $n$ elements has $C_n^2$ dependencies at most.

Definition 2 Node Dependence Value (NDV): Assuming that $T_{DS}$ is the dependence set corresponding to the test case set $TS$, then the node dependence value of $T_k$ ($T_k \in TS$) is the accumulation of the initial node’s NDV of all dependencies plus 1 (the terminal node of all dependencies being $T_k$), expressed as

$$\text{NDV}(T_k) = \sum_{c=1}^{m}[\text{NDV}(T_c) + 1]. \tag{3}$$

where, $m$ is the number of dependencies in which $T_k$ is the terminal node in $T_{DS}$. The NDV of the test cases which never appear in the terminal node in $T_{DS}$, is 0.

NDV can be used to characterize the priority of test cases. The higher the NDV, the higher the executive priority of the node. That is, the test case set arranged in reverse order according to NDV satisfies the executive priority of the test cases. It can be demonstrated that this executive sequence eliminates the impact of the dependencies between test cases. Given $\text{NDV}(T_1) \geq \text{NDV}(T_2)$ for $T_1$ and $T_2$, then executing $T_1$ before $T_2$ can eliminate the dependence, which is proved as follows:

If executing $T_1$ before $T_2$ can not eliminate the dependence, $T_2$ depends directly or indirectly on $T_1$. The NDV of $T_2$ will add a positive integer on the basis of the NDV of $T_1$, leading to $\text{NDV}(T_2) > \text{NDV}(T_1)$, and that is contradictory to the given condition.
Test Case Slice

The program slice is a program decomposition technique based on the relevant characteristics of the program, dividing the program into a number of decomposed slices which will be analyzed to understand the entire program. This section introduces the test case based on the slice theory.

Definition 3 Dependence Loop (DL): Assuming that DS is the dependence set corresponding to \( T_1 \), \( T_2 \), ..., \( T_n \), \( DS_1 \subseteq DS \), and all test cases of \( DS_1 \) makes up \( \{T_i, T_j, ..., T_k\} \) \((i, j, ..., k \in [1, 2, ..., n])\) and \( i, j, ..., k \) are different. If \( DS_1 \) satisfies the two conditions:

1. each pair of dependencies is visited only once;
2. the terminal node of each dependence is the only initial node for another dependence.

Then \( \{T_i, T_j, ..., T_k\} \) forms a dependence loop (DL), recorded as

\[
DL = < T_i, T_j, ..., T_k >.
\]

(4)

All the dependence loops of one test case set is called dependence loop set (DLS), recorded as

\[
DLS = \{L_1, L_2, ..., L_m\}.
\]

(5)

where, \( m \) is the number of dependence loops for the test case set.

The dependence loop can be constructed by referring to the generation method of the directional loops in the graph theory, the existing research ideas and solutions.

Definition 4 Test Case Slice (TCS): Test case slice is a subset defined in \( TS = \{T_1, T_2, ..., T_n\} \), and there is no dependence loops in the subset. TCS can be written as

\[
TCS = \{T_i | i = 1, 2, ..., k\}.
\]

(6)

where, \( k \) represents the number of elements in the slice, and \( k \leq n \). For a test case set, it can eventually be broken into test case slices which are called test case slice set (TCSS), recorded as

\[
TCSS = \{TCS_i | i = 1, 2, ..., m\}.
\]

(7)

where, \( m \) represents the number of test case slices, and \( m \leq n \). For any \( TCS_i \) and \( TCS_k \) \((i \neq k\) and \( i, k \in [1, 2, ..., m]\)), \( TCS_i \cap TCS_k = \emptyset \); For all \( TCS \), \( \cup_{i=1}^{m} TCS_c = TS \).

If there are dependence loops in a test case set, then all the elements in the loops will appear on the left and right sides of the NDV's equation group, and NDV is unable to be calculated. So NDV applies to the test case set having no dependence loops, and the test case slice is a special set having no loops.

Test Sequence Generation

Dependence-Slice Algorithm

Based on the idea of slicing, the test case set can be divided into several test case slices, and taking slices as the test unit, the test cases within the slice are optimized to reduce or even eliminate the impact of the test cases. This paper proposes a method of test sequence generation based on dependence and slices (TSGDS). The algorithm flow is shown in Figure 1.
From the algorithm flow of TSGDS, when Putting TS and DS as inputs, we can get TCSS that sorted by NDV and test sequence. The algorithm mainly includes three phases: the initialization, the loops cutting to obtain slices and the sorting in the slices to generate sequence. The specific process is as follows:

**Phase1. Initialization**
Define and assign the variables of the algorithm. Put test case set $ts$ and dependence set $ds$ as inputs, other variables is as follows:

- The intermediate test case set $t_{ts} = ts$, the intermediate dependence set corresponding to $t_{ts}$ $t_{ds} = ds$, the largest element set $ts_{max} = \emptyset$, the number of slices $n = 1$, the dependence loop set $dls = \emptyset$, the maximum-frequency element $T_{max} = \emptyset$, and the test case slice set $tcss = \emptyset$.

**Phase2. Loops cutting to obtain slices**
As a recursive process, the loops cutting to obtain slices is the core of the algorithm and achieves the maximum cut loops. The process is as follows:

Step1. Construct $dls$ according to $t_{ts}$ and $t_{ds}$ with the loops construction algorithm.
Step2. If $dls$ is null, break this step. If $dls$ is not null, traverse $dls$ to get the frequency of each test case, find $T_{max}$ and puts it into $ts_{max}$, then delete the loops containing $T_{max}$ from $dls$. If there are multiple largest elements, get the element that do not constitute loops with $ts_{max}$ as $T_{max}$.
Step3. If $ts_{max}$ is null, $tcs_{n} = t_{ts}$, put $tcs_{n}$ into $tcss$, and break phase2. Else, make $n$ plus one, $tcs_{n}$ is equal to $t_{ts}$ minus $ts_{max}$; Since there is no loops in $tcs_{n}$, put $tcs_{n}$ into $tcss$.
Step4. If there are no loops in $ts_{max}$, $tcs_{1} = ts_{max}$, put $tcs_{1}$ into $tcss$, clear $ts_{max}$. If not, $t_{ts} = ts_{max}$, $ts_{max} = \emptyset$, obtain $t_{ds}$ corresponding to $t_{ts}$, and then execute Step 1, Step 2 and Step 3 again.

**Phase3. Sorting in the slices to generate sequence**
Construct dependence sets $ds_{1}, ds_{2}, ..., ds_{n}$ corresponding to $tcs_{1}, tcs_{2}, ..., tcs_{n}$, obtain the NDV of each test case of each slice, and then arrange the test cases of each slice in the reverse order according to the NDV.

We can see from TSGDS that each element of $tcss$ satisfies the definition of TCS, and slices is ordered reversely by NDV, so the resetting of database is only performed between the slices and not in slices. Therefore, the number of slices $n$ and the times of resetting of database $reC$ have the following relationship: $reC = n - 1$.

**Case Study**
Assuming that $ts = \{T_1, T_2, T_3, T_4, T_5\}$, $ds = \{< T_1, T_2 >, < T_2, T_5 >, < T_5, T_1 >, < T_4, T_2 >,$
$T_3, T_1 >, < T_4, T_1 >, < T_3, T_4 >, < T_4, T_5 >, < T_2, T_3 >, < T_5, T_3 >$ . In the initialized state, $tTs = ts$, $t.dls = ds$, $ts_{max} = \emptyset$, $n = 1$, $ds = \emptyset$ and $T_{max} = \emptyset$, then the process of generating test sequence is as follows:

1) The first recursion to separate loops to get slices
   (1) Constructing $dls$, it is $\{< T_1 T_2 T_3 >, < T_1 T_2 T_3 T_4 >, < T_1 T_2 T_3 T_4 T_5 >, < T_2 T_3 T_4 >, < T_1 T_2 T_5 >, < T_1 T_2 T_5 T_3 >, < T_4 T_2 T_5 T_3 >\}$. The first time to determine that $dls$ is not empty, traverse $dls$ to obtain the frequency of each test case, then $T_{max} = T_2$; put $T_{max}$ into $ts_{max}$ and delete the loops including $T_{max}$, then $ts_{max} = \{T_2\}$, $dls = \{< T_3 T_4 T_5 >\}$. (2) The first time to determine that $dls$ is not empty, traverse $dls$ to obtain the frequency of each test case. $T_3$, $T_4$ and $T_5$ are the elements of largest frequency in parallel and have no loops with $\{T_2\}$. This case makes $T_{max} = T_3$, then $ts_{max} = \{T_2, T_3\}$, $dls = \emptyset$. (3) The second time to determine that $dls$ is not empty, traverse $dls$ to obtain the frequency of each test case. $T_3$, $T_4$ and $T_5$ are the elements of largest frequency in parallel and have no loops with $\{T_2, T_3\}$. This case makes $T_{max} = T_3$, then $ts_{max} = \{T_2, T_3\}$, $dls = \emptyset$.

2) The second recursion
Since there is no loop in $\{T_2, T_3\}$, the process of recursion is ended. $tcs_1 = ts_{max} = \{T_2, T_3\}$, put into $tcs_1$ into $tcs$, so $tcss = \{tcs_1, tcs_2\}$.

3) Sorting in the slice to generate sequence
   (1) Sort reversely the test cases in the slices according to the NDV. For $tcs_1 = \{T_2, T_3\}$, $ds_1 = \{< T_2, T_3 >\}$, then:
   \[
   NDV(T_3) = NDV(T_2) + 1 \Rightarrow \begin{cases} NDV(T_2) = 0 \\
   NDV(T_3) = 1 \end{cases} \Rightarrow tcs_1 = \{T_3, T_2\}
   \]

   For $tcs_2 = \{T_1, T_4, T_3\}$, $ds_2 = \{< T_5, T_1 >, < T_4, T_1 >, < T_4, T_5 >\}$, then:
   \[
   \begin{cases} NDV(T_1) = NDV(T_5) + 1 \\
   NDV(T_4) = 0 \end{cases} \Rightarrow \begin{cases} NDV(T_1) = NDV(T_5) + 1 \\
   NDV(T_4) = 1 \end{cases} \Rightarrow tcs_2 = \{T_1, T_5, T_4\}
   \]

   (2) Generating the test sequence: The test sequence is $T_1, T_5, T_4, T_3, T_2$ or $T_3, T_2, T_1, T_5, T_4$.

Algorithm Analysis and Experiments

Algorithm Complexity

According to the algorithm flow, the algorithm cost is mainly reflected in the process of slicing and sorting. The Fast Sorting is used for the sorting of slices and the algorithm complexity is $O(n\log n)$. We can analyze the process of slicing through recursion. Assuming that the length of the test case set and dependence set are $n$ and $m$ respectively, the number of slices is $k$, then the complexity of the algorithm is as follows:

1) In the best case. $k = 1$, i.e. no loops exist in the test case set, and the time complexity is determined by the ordering in the slices.

2) In the worst case, the time complexity is determined by slicing, which is a recursive process, we can get time complexity according to recursive equation, then time complexity is $O(n^2)$.

Table 1 compares the time complexity with the results in the 13th literature. Table 1 exhibits that the time complexity of TSGDS is basically at the same grade as that of Literature 13, especially for the test case set with dependence loops.

<table>
<thead>
<tr>
<th>Programs</th>
<th>Best case</th>
<th>Worst case</th>
</tr>
</thead>
<tbody>
<tr>
<td>Literature 13</td>
<td>$O(n + m)$</td>
<td>$O(n^2)$</td>
</tr>
<tr>
<td>TSGDS</td>
<td>$O(n\log n)$</td>
<td>$O(n^2)$</td>
</tr>
</tbody>
</table>

Experimental Comparisons

During the execution of the test cases, the time of the resetting of database is much greater than that of the execution of test cases, so the efficiency of the test sequence is characterized by the number
of resetting of database. The Experiments selected the following two groups of data:

1) Typical test data
Typical test data \([12,13]\) includes Example 1, Example 2 and Example 3, as shown below:
Example 1: \(ts = \{T_1, T_2, T_3, T_4, T_5\} \), \(ds = \{< T_1, T_2 >, < T_2, T_5 >, < T_5, T_1 >, < T_4, T_2 >, < T_3, T_1 >, < T_4, T_3 >, < T_5, T_3 >\}\).
Example 2: \(ts = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}\} \), \(ds = \{< T_1, T_2 >, < T_3, T_2 >, < T_2, T_4 >, < T_4, T_3 >, < T_5, T_4 >, < T_6, T_5 >, < T_7, T_6 >, < T_8, T_7 >, < T_9, T_8 >, < T_{10}, T_9 >\}\).
Example 3: \(ts = \{T_1, T_2, T_3, T_4, T_5, T_6, T_7, T_8, T_9, T_{10}, T_{11}\} \), \(ds = \{< T_1, T_2 >, < T_1, T_8 >, < T_4, T_1 >, < T_2, T_4 >, < T_2, T_5 >, < T_2, T_3 >, < T_3, T_4 >, < T_3, T_5 >, < T_6, T_3 >, < T_5, T_6 >, < T_6, T_7 >, < T_9, T_7 >, < T_9, T_{10} >, < T_7, T_8 >, < T_{10}, T_8 >, < T_8, T_9 >, < T_8, T_{11} >, < T_{11}, T_{10} >, < T_6, T_9 >, < T_8, T_3 >, < T_3, T_1 >\}\).

2) Test data of Halo Convenience Store Management System
The system includes four functional modules: warehouse management, query management, store settings, bulletin management. The number of test cases in the system is more than 300, including 113 cases for warehouse management, 48 cases for query management, 62 cases for store settings, 35 cases for bulletin management and 57 cases for report analysis.

For the experimental data, we chose 4 sequences for comparisons. The experimental results are shown in Figure 2. It can be seen from Figure 2 that the number of resetting of database for TSGDS is smaller than order sequence, reverse sequence, random sequence and the sequence generated from Literature 13.

![Figure 2. The number of resetting of database for typical case set and Halo system.](image)

Conclusion
On the basis of the existing optimization technology of the test sequence, this paper analyzes the influence of dependencies on the execution of test cases, proposes a method to generate test sequence based on dependencies and slices. TSGDS includes three phases, eliminates the test effect caused by dependencies, improves the efficiency of the test cases. The deficiencies of this method are that recursive algorithm used in the process of separating loops increases the complexity, and the inter-loop-nest are not fully considered in constructing loops. Further research is needed In the future.

Acknowledgments
This project is sponsored by “the Scientific Research Foundation for the Overseas Chinese Scholars.”

References


