A HYBRID DYNAMIC PROGRAM SLICING

Yi Tong  Wu Fangjun

(Laboratory of Computer Center, Yichun University, Yichun 336000, China)

Abstract This letter proposes a hybrid method for computing dynamic program slicing. The key element is to construct a Coverage-Testing-based Dynamic Dependence Graph (CTDDG), which makes use of both dynamic and static information to get execution status. The approach overcomes the limitations of previous dynamic slicing methods, which have to redo slicing if slice criterion changes.

Key words Program analysis; Program slicing; Dynamic slicing; Software testing

I. Introduction

Dynamic slicing is an important technology in software engineering, program understanding, maintenance, debugging, testing, differencing, specialization reuse, optimization, and so on[1,2]. There are many approaches to dynamic slicing[2-5]. Their major shortcoming is either imprecision or inefficiency. Furthermore, if slice criterion changes, most approaches have to redo the slicing. However, in practice, dynamic slicing has to be done repeatedly to find as many errors as possible in software. Therefore, it is highly necessary to introduce an effective dynamic slicing method to overcome the limitations of previous methods and reduce blindness.

In our previous work, we have done some research in dependence analysis[6,7]. This letter extends previous work to dynamic slicing, and proposes a Coverage-Testing-based Dynamic Dependence Graph (CTDDG). Firstly, partial paths of a program are selected as testing paths. Secondly, a CTDDG is constructed after analyzing dependences between nodes. Thirdly, a new dynamic slicing method is given. Finally, the advantages of the proposed method are demonstrated.

II. Dynamic Slicing Based on CTDDG

In order to manage an exhaustive test, every possible path must be executed for at least one time. It is very hard to accomplish even for a small program. Therefore, it is crucial to select typical and detecting-easy error execution paths in the process of testing. Generally, a program contains three structures: sequential structure, loop structure, and branch structure. So we emphase on programs that contain sequential structure, loop structure, and branch structure. Transfer structure can be managed similarly. For the combination of sequential and branch structures, the result is the same as that analyzing the two structures separately. For the combination of branch and loop structures, we define a longest testing path. Although different program languages have different branch structures and loop structures, essentially they are the same. So we select if-then-else, while <conditions> do and

---

1Manuscript received date: April 2, 2004; revised date: May 20, 2004.
Communication author: Yi Tong, born in 1975, male, Ph.D. Center of Modern Education at Yichun University, Yichun 336000. China. tongyi@seu.edu.cn
for <conditions> do as typical structures for analysis. For the convenience of discussion, we number program structures ascendingly. The beginning number is 1. Some relevant concepts to CTDDG will be defined as follows.

**Definition 1** Let $s_m, s_n, s_t$ be structures of program $P$, where $m \leq n \leq t$. Then,

1. $s_m$ represents a condition structure if <conditions>, $s_n$ represents the last structure of the then-branch, and if sequence structures $<s_m s_{m+1} \cdots s_n>$ is a path, then it is called a condition path, denoted by $\text{con-prn}$. If the else-branch is nonempty, then let $s_t$ represent the last structure of the else-branch; if the sequence structures $<s_m s_{n+1} \cdots s_t>$ is a path, then it is called a condition path, denoted by $\text{con-pnt}$. Otherwise, if the else-branch is empty, and if $s_n$ is not the last structure of program $P$, then sequence statements $<s_m s_{n+1}>$ is a condition path, denoted by $\text{con-pro,n+1}$; if $s_n$ is the last structure of program $P$, then $<s_m>$ is called a condition path, denoted by $\text{con-pro, end}$.

2. $s_m$ represents a loop structure, $s_n$ represents the last structure of inner loop. If loop structure is not the last structure of program $P$, then $s_t$ represents the first structure of outer loop. If $<s_m s_{m+1} \cdots s_n>$ is a path, then it is called a loop path, denoted by $\text{loop-prn}$. And if $<s_m s_t>$ is a path, then it is called a loop path, denoted by $\text{loop-pnt}$.

3. Both condition path and loop path from $s_m$ to $s_n$ are called testing path, denoted by $\text{Pron}$.

4. If $p_{mn}$ represents a testing path, $p_{kt}$ is any other testing path, for $p_{mn} \neq p_{kt}$, where $k < m \leq n \leq t$, then $p_{mn}$ is called a longest testing path. Corresponding node to $s_i$ in the longest testing path $p_{mn}$ is called a testing node, denoted by $s_i^{p_{mn}}$, otherwise called non-testing node. Specially, we distinguish a structure from its corresponding node only if there is a testing path.

If more than one longest testing paths have the same beginning node and end node, we append a suffix to the end node to distinguish them. It should be emphasized that testing paths are different from traditional executable paths.

After selecting some partial paths as testing paths, we can analyze relations between testing nodes and non-testing nodes. The relations are defined as three dependences: Data, control and same-structure dependences.

**Definition 2** Let $s_i$ and $s_j$ be structures of program $P$, where $i \leq j$. If a variable $v$ exists, and all the below conditions hold, then $s_j$ is directly data dependent on $s_i$, denoted by $P_{\text{DEPd}}(s_i, s_j)$:

1. $s_i$ defines $v$;
2. $s_j$ uses the value of $v$ in execution process;
3. (a) There is a longest testing path $p_{mn} = s_m \cdots s_i \cdots s_k \cdots s_j \cdots s_n$ to which both $s_i$ and $s_j$ belong. For any structure $s_k$ in $p_{mn}$, $s_k$ does not redefine $v$, where $m \leq i < k < j \leq n$; or

(b) There is not any longest testing path $p_{mn}$ to which both $s_i$ and $s_j$ belong, but for any path=$<s_i \cdots s_j>$, in which there is not any other structure between $s_i$ and $s_j$ that redefines $v$, where $m \leq i < j \leq n$.

**Definition 3** Let $s_i$ and $s_j$ be structures of program $P$, where $i \leq j$, data dependence $P_{\text{DEPd}}^*(s_i, s_j)$ is satisfied only if one of following holds:

1. $P_{\text{DEPd}}(s_i, s_j)$;
2. $\exists s_k (P_{\text{DEPd}}(s_i, s_k) \land P_{\text{DEPd}}^*(s_k, s_j))$