AN INTERACTIVE DEBUGGING TOOL FOR C++
BASED ON DYNAMIC SLICING AND DICING
PART I: DEFINITIONS AND ALGORITHMS

Winai Wichaiapanitch¹ and M. H. Samadzadeh²

Abstract

The main objective of this work was to develop an interactive debugging tool for C++ programs. The tool that was developed is called C++Debug and it uses program slicing and dicing techniques. The design started by including simple statements first and then expanded to pointers, structures, functions, and classes. In order for C++Debug to be more powerful, dynamic slicing rather than static slicing was chosen. The work includes new algorithms that handle Class, Function, and Pointer in C++.

The results of this work are reported in two parts:
PART I: Definitions and Algorithms,
PART II: Implementation, Testing, and Evaluation.

This is part I that reports on the definitions and algorithms, how to compute a slice, and the dicing procedure.

1. Introduction

Since the article “Program Slicing” by Mark Weiser was initially published in 1981 [10], program slicing has gained wide recognition in both academic and practical arenas. Several debugging tools have been developed that utilize program slicing. For example, Focus [6] (designed and implemented by Lyle in 1984) was designed to be used with Fortran programs, and C–Slicer [7][8] (designed and implemented by Nanja and Samadzadeh in 1990) and C–Debug [9] (designed and implemented by Wichaiapanitch and Samadzadeh in 1992) were designed to be applicable to C language programs based on dynamic slicing. Program slicing [1][2][10][11][12][13] is one of the debugging methods used to localize errors in a program. The idea of program slicing is to focus on the statements that have something to do with a certain variable of interest (criterion variable), with the unrelated statements being omitted. Using slicing, one obtains a new program of generally smaller size that still maintains all aspects of the original program’s behavior with respect to the criterion variable. Dynamic slicing differs from static slicing in that it is defined on the basis of a computation or an execution rather than on all possible computations. Furthermore, it allows one to treat the elements and fields in dynamic records as individual variables [5]. As a result, the slice size computed based on the dynamic slicing technique is generally smaller. Moreover, dynamic slicing allows one to keep track of the run–time

¹ On leave from Computer Engineering Department, Faculty of Engineering, Rajamangala Institute of Technology, Klong 6, Pratumthani, Thailand.
² Oklahoma State University, Computer Science Department, 312 MSCS Stillwater, OK 74078. samad@cs.okstate.edu.
type binding (involving the type of each object) that is unknown at compile time but is determined when the program is executed. Dynamic slicing technique was used in this study. Dicing technique [6][7][8] can then be used to compare two or more slices resulting from the program slicing technique in order to identify the set of statements that are likely to contain an error. The formal model of static/dynamic slicing/dicing is presented. There is a need for debugging tools that are capable of making some deductions regarding the presence and location of errors in programs.

2. Definitions

A number of definitions and algorithms originally introduced by Korel and Laski [3][4][5] were modified, in order to compute slices in classes, objects, arrays, pointers, references, dynamic allocation operators, function overloading, copy constructors, default arguments, operator overloading, inheritance, virtual functions, polymorphism, templates, and exception handling of a C++ program. Those modified definitions plus a number of new definitions and algorithms are introduced in this chapter.

Based on Korel and Laski’s work [5], let the flow graph of a program P be a directed graph (N,A,s,e) and C be a slicing criterion, where N is a set of nodes, A is a binary relation on N (a subset of N x N) referred to as the set of arcs, s ∈ N is a unique entry node, and e ∈ N is a unique exit node.

Each node in N consists of one statement, including a single instruction, a control instruction, and a function instruction. A single instruction can be, for example, an assignment statement or an input or output statement. A control instruction can be such statements as an if-then-else statement or a while statement, which are also called test instructions. A function instruction can be either a called or a calling function instruction.

An arc(n, m) ∈ A corresponds to a possible transfer of control from instruction n to instruction m.

A path from the entry node s to some node k, k ∈ N, is called a sequence <n₁, n₂, ..., n_q> of instructions, such that n₁ = s, n_q = k, and (nᵢ, nᵢ₊₁) ∈ A, for all nᵢ, 1 ≤ i < q. If there are input data that cause a path to be traversed during program execution, the path is feasible. A feasible path that has actually been executed for some input is called a trajectory.

**Definition 1.** Let X be an instruction in a program and X ∈ IN* (the set of non-negative integers). Let P be the set of instruction numbers in a tested C++ program, then P = {1, 2, ..., n} represents a program of length n, where n is the size of the program

\[ P = \{ X \mid \text{for all } X \text{ with } 1 \leq X \leq n \} \]

where n = length of the program.

**Definition 2.** Let F_{name} be a function, i.e., a set of instruction X’s in the scope of influence of the function name, where all blank lines are ignored. F_{name} ⊆ P, and F_{name} = F_{main} if the program has one function.

\[ F_{name} = \{ X \mid \text{for all } X \text{ with } i \leq X \leq k \} \]

where

1. i is the starting line number of function name, i ∈ P
2. k is the ending line number of function name, k ∈ P
Definition 3. Let $T$ be a trajectory, i.e., a feasible path that has actually been executed for some input [5]. A trajectory of length $m$ is denoted by a list $T = \langle X_1, X_2, ..., X_m \rangle$, where $X$ is an instruction of a tested C++ program.

$$T = \langle X \rangle \text{ for all } X_i \text{ where } X_i's \text{ are in a feasible path executed for some input and } X \in P >$$

Definition 4. Let $TF_{name}$ be a function trajectory, i.e., a feasible path of a function name that has actually been executed for some input. $TF_{name}$ is a sublist of $T$. If a trajectory of length $m$ is denoted by $T = \langle X_1, X_2, ..., X_m \rangle$, then the function trajectory name is denoted by $TF_{name} = \langle X_i, X_{i+1}, ..., X_k \rangle$, where $X_i, X_{i+1}, ..., X_k$ are a list of the instruction $X$'s which are in the scope of a given function $F_{name}$, where $i$ denotes the position of entry node and $k$ denotes the position of ending node of the function name, $(1 \leq i < k, \text{ and } i < k < m)$.

$$TF_{name} = \langle X \rangle \text{ for all } X_i \text{ where } X_i's \text{ are in a feasible path executed for some input, } X \in F_{name}, \text{ and } X \in T >$$

Definition 5. Let action be pair($X, p$), i.e., instruction $X$ at position $p$, which will be replaced by $X'$ for brevity and ease of understanding [5]. An action $X'$ is a test action if $X$ is a test instruction such as while or for.

Definition 6. Let $M(T)$ be a set of actions in a given trajectory $T$, where $M(T) = \{ X' : \text{instruction } X \text{ at position } p \text{ in trajectory } T \} [5]$.

Definition 7. Let $M(TF_{name})$ be a set of actions in a given function of a given trajectory $TF_{name}$, where $M(TF_{name}) = \{ X' : \text{instruction } X \text{ at position } p \text{ in trajectory } TF_{name} \}$. $M(TF_{name})$ is a subset of $M(T)$.

Definition 8. Let $C$ be a slicing criterion, which is the specification for a particular behavior of interest. A slicing criterion can be expressed as the values of some set of variables at some set of statements [10]. If we let $T$ be the trajectory of program $P$ on input $x$, a slicing criterion of program $P$ executed on $x$ can be defined as a triple $C = (x, I^l, V)$, where $I^l$ is an action in $T$ and $V$ is a subset of variables in $P$ [5].

Definition 9. Let $D(X^p)$ be the set of variables that are defined in action $X^p$, where $X^p \in M(T)$.

Let $DF_{name}(X^p)$ be the set of variables that are defined in action $X^p$, where $X^p \in M(TF_{name})$.

Definition 10. Let $U(X^p)$ be the set of variables that are used in action $X^p$, where $X^p \in M(T)$.

Let $UF_{name}(X^p)$ be the set of variables that are used in action $X^p$, where $X^p \in M(TF_{name})$.

Definition 11. Let $LF_{name}(X^p)$ be a set of variables and C++ preprocessors that are declared as a local declaration in function name.

Definition 12. Let $DU$ be a Definition–Use Relation, a relation in which one action assigns a value to an item of data and the other action uses that value [5]. Instead of using $M(T)$ as Korel and Laski did, $M(TF_{name})$ was used in this work in order to compute a slice from functions or classes.

Let $M(TF_{name})$ be a set of actions in a given trajectory $TF_{name}$, $DU_{name}$, a Definition–Use–Function Relation, is a binary relation on $M(TF_{name})$ defined as follows:

$$\text{Let } TF_{name} = \langle X_1, X_{i+1}, ..., X_j, ..., X_k \rangle, \quad \quad X^p \text{ DUF }_{name} Y^q, \quad i \leq p < t, \text{ iff there exists a variable v}$$
such that (1) $v \in UF_{name}(Y)$, and 
(2) $X^p$ is the last definition of $v$ at $t$.

where, the last definition $X^p$ of variable $v$ at $t$ is the action which last assigned a value to $v$ when $t$ was reached on trajectory $TF_{name}$.

**Definition 13.** Let LDR be a Local–Declaration Relation, a relation in which one action declares a variable and the other action defines or uses that variable.

Let $M(TF_{name})$ be a set of actions in a given trajectory $TF_{name}$. LDRF, a Local–Declaration Relation, is a binary relation on $M(TF_{name})$ defined as follows:

Let $TF_{name} = \langle X_1, X_{1+1}, \ldots, X_t, \ldots, X_k \rangle$,

$X^p$, LDRF $Y_i$, $i \leq p < t$, iff there exists a variable $v$ such that (1) $v \in UF_{name}(Y^i) \cup DF_{name}(Y^i)$, and (2) $X^p$ is the action where variable $v$ was declared in trajectory $TF_{name}$.

**Definition 14.** Let TC be a Test–Control Relation, capturing the effect between test actions and actions that have been chosen to execute by these test actions [5]. Instead of using $M(T)$ as Korel and Laski did, $M(TF_{name})$ was used in this work in order to compute a slice from functions or classes. Let $M(TF_{name})$ be a set of actions in a given trajectory $TF_{name}$. TCF, a Test–Control–Function Relation, is a binary relation on $M(TF_{name})$ defined as follows:

Let $TF_{name} = \langle X_1, X_{1+1}, \ldots, X_t, \ldots, X_k \rangle$,

$X^p$, TCF $Y^i$, $i \leq p < t$, iff 

(1) $Y$ is in the scope of influence of $X$, and 
(2) for all $k$, $p < k < t$, $T(k) \neq X$

where, the scope of influence is defined as follows.

(1) If $X$ then B1 else B2; Instruction $Y$ is in the scope of influence of $X$ iff $Y$ is in B1 or B2.

(2) While $X$ do B; Instruction $Y$ is in the scope of influence of $X$ iff $Y$ is in B.

(3) Do B while $X$; Instruction $Y$ is in the scope of influence of $X$ iff $Y$ is in B.

(4) Case $X$ do B; Instruction $Y$ is in the scope of influence of $X$ iff $Y$ is in B.

(5) For $X$ do B; Instruction $Y$ is in the scope of influence of $X$ iff $Y$ is in B.

(6) Function $X$ do B; Instruction $Y$ is in the scope of influence of $X$ iff $Y$ is in B.

**Definition 15.** Let IRF be an Identity Relation in Function, then $X^p$, IRF $Y^i$, iff $X = Y$ is the identity relation IRF on $M(Front(TF_{name}, q))$, where $Front(TF_{name}, q)$ is a sublist of $TF_{name}$ consisting of the first $q$ elements of $TF_{name}$, where $TF_{name} = \langle X_1, X_{1+1}, \ldots, X_{t_1}, \ldots, X_{q}, \ldots, X_k \rangle$ denotes a function trajectory, $q$ is a position in $TF_{name}$, $1 \leq i < t$, and $t < q \leq k$.

**Definition 16.** Figures 1 and 2 present a part of the trajectory of FuncA(int i) and FuncB(int j), where called FuncA(int i) is called by calling FuncA(5) at $X^{i+1}$, and called FuncB(int j) is called by calling FuncB(2) at $X^{i+1}$. From Figures, we find that $T = \langle X^{i-2}, X^{i-1}, X^i, X^{i+1}, X^{i+2}, \ldots, X^j, X^{j+1}, \ldots, X^k, X^{k+1}, \ldots, X^l, \ldots, X^m, \ldots, X^n, X^{n+1}, X^{n+2}, \ldots, \rangle$, where $i < j < k$, $l \leq m$ and $X$ is any statement in a program $P$, $TF_{FuncA} = \langle X^1, \ldots, X^l \rangle$. 


from a called action first and then from a calling action. For example, in Figure 1, suppose one needs to find a slice of variable U at \( X^k \). The process starts from \( X^k \) (which is in the scope of influence of called function FuncB(int j), which is called by calling function FuncB(2) at \( X^{i+1} \)), and then \( X^{i+1}, X^{i+1} \), respectively. We find that called action \( X^{i+1} \) comes before calling action \( X^{i+1} \).

Calling→Called occurs when a slice is computed from a calling action first and then from a called action. For example, in Figure 2, suppose that one needs to find a slice of variable Z at \( X^m \). The process starts from \( X^m \), and then \( X^{i+1} \) (since Z is last defined at \( X^{i+1} \) and used at \( X^m \)) and then \( X^{i+1} \) (since called FuncB(int j) is called by calling FuncB(2)), respectively. We find that calling action \( X^{i+1} \) comes before called action \( X^{i+1} \).

Modified from Korel and Laski’s approach [5], let \( \text{TF}_{\text{name}} = \langle X, X_{i+1}, X_{i+2}, ..., X_k \rangle \) be a trajectory of function name, and q be a position in \( \text{TF}_{\text{name}} \) where \( i,q \leq k \). Then \( \text{Front}(\text{TF}_{\text{name}}, q) \) is a sublist \( \langle X_{i+1}, ..., X_k \rangle \) and \( \text{Back}(\text{TF}_{\text{name}}, q) \) is a sublist \( \langle X_{i+1}, ..., X_k \rangle \) as shown in Figures 1 and 2. All Back(\( \text{TF}_{\text{name}}, q) \)’s can be ignored in computing a slice. Just Front(\( \text{TF}_{\text{name}}, q) \) must be concentrated on.

Let A and B be two functions, where function A calls function B. Therefore, a slice can be computed in two different ways as follow.

1) Called→Calling

Total slice \( AB = \text{Slice}_B \cup \text{Slice}_A \)

where

\( \text{(1) Slice}_B \) is a slice computed based on \( \text{Front}(\text{TF}_B, k) \) and slicing criterion \( C = (x, X^q, V) \)

\( \text{(2) Slice}_A \) is a slice computed based on \( \text{Front}(\text{TF}_A, 1+1) \) and used variables at calling action \( X^{i+1}, U(X^{i+1}) \).
2) Calling-to-Called

Total slice $\text{Slice}_{AB} = \text{Slice}_A \cup \text{TF}_B$

where

(1) Slice $\text{Slice}_A$ is a slice computed based on
Front($\text{TF}_A, m$) and slicing criterion $C = (x, x^m, V)$,
(2) $\text{TF}_B$ is a function trajectory of function $B$.

Let Calling($X^p$) be a set of calling functions that are used to call a called function in action $X^p$, where $X^p \in M(T)$.

Let Called($X^p$) be a set of called functions that are called by a calling function in action $X^p$, where $X^p \in M(T)$.

Let EI be a Called-to-Called Relation between called and calling functions. Let $M(T)$ be the set of actions in a given trajectory $T$ of length $m$. 
EI is a binary relation on $M(T)$ defined as follows:

Let $T = \langle X_1, X_2, \ldots, X_1, \ldots, X_m \rangle$

such that

(1) a called function $f \in \text{Called}(Y^t)$,
(2) a calling function $f \in \text{Calling}(X^p)$,
(3) $X^p$ is the calling action, where the calling function $f$ at $p$ calls a called function $f$ at $t$.

Let IE be a Calling-to-Called Relation between called and calling functions.

Definition 17. To find the slicing set $S_n$, we first find the set $A^0$ of all actions that have direct influence on $V$ at $q$ and on action $I^r$. $A^0$ is defined as follows [5].

$A^0 = \text{LD}(q, V) \cup \text{LT}(I^r) \cup I^r$

where $\text{LD}(q, V)$ is the set of last definitions of variables in $V$ at the execution position $q$, and $\text{LT}(I^r)$ is a set of test actions that have Test-Control influence on $I^r$.

We will find $S_n$ iteratively, as the limit of a sequence $S_0, S_1, \ldots, S_n, 0 \leq n < q$, which is defined as follows.

$S_0 = A^0$ and $S_n+1 = S_n \cup A^{n+1}$

where

(1) $X^p \in S_n$,
(2) there exists $Y^t \in S_i, t < q, X^p Z Y^t$

Finally, we can get the slice from the following definition.

$S_i = S^k$

where $S^k$ is the limit of the sequence $\{S_i\}$.

Definition 18. Let FN(q) be a string of function name such that $X^q$, $X$ is in the scope of influence.

Definition 19. Let $G(X)$ be a set of variables and preprocessors that are declared as a part of global declaration. $G(X)$ is computed from the source program, not from a trajectory path.

Definition 20. Let VDU(FunctionName) be a set of variables that are used, UF name, and defined, DF name, in a given function name.
Definition 2.1. In order to find the scope of influence of each instruction, variable scope, VS, and control scope, CS, are used as defined below.

1. Variable scope, VS, gives the information that variables that used or defined in each instruction were declared at what instructions.

Let \( X_{\text{dcl}} \) be an instruction that declared variables such as “int I;”.

Let \( X_{\text{du}} \) be an instruction that used or defined the variables declared by \( X_{\text{dcl}} \), where variables that are used or defined are in the scope of influence of the variables that are declared in \( X_{\text{dcl}} \). For example, “I=I+1;”, which is the first I is defined and the second I is used both are declared by “int I;”.

Then we get \( \text{VS}(X_{\text{du}}) \), a variable scope relation at \( X_{\text{du}} \), which is a set of instructions \( X_{\text{dcl}} \), where \( X_{\text{du}} \) is in the scope of influence of \( X_{\text{dcl}} \).

2. Control scope, CS, gives information about instructions that are in the scope of influence of control instructions such as test statements, functions, and classes. For calculation of the scope of influence of each statement, the me_too set is used [6].

Let X be an instruction, the me_too is a set of instructions that are in the scope of influence of instruction X.

Due to the complexity of the C++ language and in order for C++Debug to be applicable to programs containing functions, classes, namespaces, unions, structures, and preprocessors (a separate first step in compilation, e.g., #include, #define, or #if), the me_too set was modified according to the rules shown in Fig. 4.

<table>
<thead>
<tr>
<th>Instruction (X)</th>
<th>Prototype</th>
<th>Called</th>
<th>Calling</th>
<th>D</th>
<th>U</th>
<th>DCL</th>
<th>VS</th>
<th>CS</th>
</tr>
</thead>
<tbody>
<tr>
<td>include &lt;iostream&gt;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>26</td>
<td></td>
</tr>
<tr>
<td>1: class Compute {</td>
<td>Compute(01)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>private;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2: int Max;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>3: float Num[4];</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>4: public;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>5: Compute(int M, float *N)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>6: Max = M;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>7: cout&lt;&lt;&quot;allocate mem&quot;&lt;&lt;endl;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>8: for(int I = 0; I&lt;=Max; ++I)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>9: Num[I] = N[I];</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>10: }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>11: }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>12: float Sum(void)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>13: float Tsum = 0;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>14: for(int I = 0; I&lt;=Max; ++I)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>15: Tsum = Tsum + Num[I];</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>16: }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>17: return Tsum;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>18: main()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>19: int Max = 4;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>21: Compute(Max, Num);</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>22: cout&lt;&lt;A Sum()&lt;&lt;endl;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>23: cout&lt;&lt;A Avg()&lt;&lt;endl;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>24: }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>25: }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>26: }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>27: }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>28: }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>29: int Max = 4;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>31: Compute(Max, Num);</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>32: cout&lt;&lt;A Sum()&lt;&lt;endl;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>33: cout&lt;&lt;A Avg()&lt;&lt;endl;</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>34: }</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>3</td>
<td></td>
</tr>
</tbody>
</table>

Fig. 3 The Prototype, Called, Calling, D, U, DCL, VS, and CS sets for the program depicted in Fig. 8
and will still be called the control scope, CS, set.

Based on the rules in Figure 3, Figures 4 shows an example of computing the CS set of a tested program that computes the sum and average of a set of numbers in Figure 8.

To find the final slicing set $F_s$ with scope, we first find the set $S^0$ of all instructions that sliced from the tested program $P$ based on slicing criterion $C(x, F, V)$. $S^0$ is defined as follows.

$$S^0 = S_c$$

where $S_c$ is a slicing set defined in Definition 17.

We will find $F_s$ iteratively, as the limit of a sequence $F^0, F^1, ..., F^n, 0 \leq i \leq n, n =$ length of program $P$, which is defined as follows.

$$F^0 = S^0 \text{ and } F^{i+1} = F^i \cup S^{i+1}$$

where

$$S^{i+1} = \{ X \in P : 1 \leq X < n, n = \text{ length of program } P, X \in F^i, \text{ and } (1) \ X \in F_s, \ (2) \ there \ exists \ Y \in F^i, X \in Z(Y) \}$$

where $Z = VS \in CS$

Finally, we can get the final slice with scope from the following definition.

$$F_s = F^k$$

where $F^k$ is the limit of the sequence $\{F^i\}$.
Fig. 5 Algorithm to compute a set of slices

![Image of algorithm from the text]

**Fig. 5 Algorithm to compute a set of slices**

Function `compute_slice_in_function_name(sliceCriterion C)`

- Get function name, see Definition 18: `name = FNC(C[n])`.
- Compute sublist function trajectory, see Definition 4: `TName = Subf(U(C))`.
- Compute defined var., see Definition 9: `DFName = ComputeDFName(TName)`.
- Compute used var., see Definition 10: `UName = ComputeUName(TName)`.
- Compute declared rel., see Def. 12: `DUFName = ComputeDUFName(TName)`.
- Compute calling-local declaration relation see Definition 13: `TCFName = ComputeTCCFName(TName)`.
- Compute identity rel., see Def. 15: `IRFName = ComputeIRFName(TName)`.
- Compute local declaration relation see Definition 16: `LDFName = ComputeLDFName(TName)`.

Function `compute_slice_in_function_name(sliceCriterion C)`

- Get calling-to-called function name: `name = FNC(p)`.
- Get calling function name: `s = SUTEName`.
- Where `IE`, a calling-to-called function.
- Is an element of `S`.
- Return `S`.

**Fig. 6 Algorithm to compute a slice of each function**

Function `main()`. Compute `scope_of_influence`

```plaintext
// function to compute the scope of influence of a slice
Add_Scope_of_Influence(array[1..n] of section) S { // Add scope of influence to a slice,
    S = Var_Control_Scope(S); // see Definition 21
    return S;
}
```

**Fig. 7 Function to compute the scope of influence of a slice**

(C) makes the final slice completed by adding some statements that may govern each statement in the slice.

**4. Examples: How to Compute a Slice**

The program in Figure 8 computes the sum and average of integers. In this example, variable Max is 4 and the array called Num contains 10.0, 20.0, 15.0, and 5.0. Upon completion of program execution, the program should yield one results as 12.5. However, this program contains an error in line 24. Rather than return `Sum()/Max`, the program computes return `Sum()/(Max + 1)`, thus yielding an error (Avg = 10.0 instead of 12.5). To localize such an error, program slicing and dicing techniques can be used. The trajectory of the program in Figure 8 is shown in Figure 9.

**Example 1.** Consider trajectory T in Figure 9. Using the criterion $C = (x, \text{Max}^\text{Avg})$, we have $x = (\text{Max}, \text{Num}) = (3, (10.0, 20.0, 15.0, 5.0))$.

The step-by-step trace of the algorithm in Figure 5.
19. return Tsum;
20. }
21. }
22. 
23. float Avg(v, y) {
24. 
25. 
26. }
27. 
28. main() {
29. int Max = 4;
30. float Num[4] = {10.0, 20.0, 15.0, 5.0};
31. Compute A(Max, Num);
32. cout << "Avg: " << endl;
33. cout << "S[0] = " << S[0] << endl;
34. 

Fig. 8 A program for calculating the sum and average of a set of numbers

follows.

Step 1:

\[ S[0] = S[0] \cup \text{Compute Slice in Function Name}(C) \]

Step 1.1:

\[ \text{FN}(C, q) = \text{FN}(30) \text{ "main"} \]

// therefore compute slice in function "main"

compute TF

// as shown in Figure 9

compute LDF

// as shown in Figure 10

compute DUF

// as shown in Figure 10

compute TCF

// as shown in Figure 10

DUF_M = \{ \}

LDF_M = \{ \}

IRF_M = \{ \}

TF_M = \{ \}

Fig. 10 The DUF_M, TCF_M, LDF_M, and IRF_M relations that are called by 32\text{th} for the trajectory depicted in Figure 9

DUF_S = \{ \}

LDF_S = \{ \}

IRF_S = \{ \}

TF_S = \{ \}

Fig. 11 The DUF_S, TCF_S, LDF_S, and IRF_S relations that are called by 32\text{nd} for the trajectory depicted in Figure 9

DUF_S = \{ \}

LDF_S = \{ \}

IRF_S = \{ \}

TF_S = \{ \}

Fig. 12 The DUF_S, TCF_S, LDF_S, and IRF_S relations that are called by 32\text{rd} for the trajectory depicted in Figure 9

DUF_S = \{ \}

LDF_S = \{ \}

IRF_S = \{ \}

TF_S = \{ \}

Fig. 13 The DUF_S, TCF_S, LDF_S, and IRF_S relations that are called by 32\text{th} for the trajectory depicted in Figure 9

compute IRF

// as shown in Figure 10

compute LDF

// as shown in Figure 10

Fig. 9 The trajectory of the program from Figure 8 on input data Max = 4, Num = (10.0, 20.0, 15.0, 5.0)
Step 1.2:

\[ S = \text{ComputeSlice} (\text{DUF}_{\text{main}}, \text{TCF}_{\text{main}}, \text{IRF}_{\text{main}}, \text{LDRF}_{\text{main}}, C) \]

Since \( C = (x, 33^{30}, \{ \text{Avg} \}) \) // given

LD(30, \{Avg\}) = \{ \}, LT(33^{30}) = \{28^{1}, 1^{11} \}, I' = 33^{30}

\[ A^0 = \{28^{1}, 33^{30}\}, \quad S^0 = \{28^{1}, 33^{30}\}, \]
\[ A^1 = \{31^{12}\}, \quad S^1 = \{28^{1}, 31^{12}, 33^{30}\}, \]
\[ A^2 = \{29^{2}, 30^{3}\}, \quad S^2 = \{28^{1}, 29^{2}, 30^{3}, 31^{12}, 33^{30}\}, \]
\[ A^3 = \{ \}, \quad S^3 = \{28^{1}, 29^{2}, 30^{3}, 31^{12}, 33^{30}\}, \]
\[ S_e = S^3 = \{28^{1}, 29^{2}, 30^{3}, 31^{12}, 33^{30}\} \]

**Step 1.3:** Check Calling-to-Calculated functions

Yes, since \( \{23^{21}\} \) IE \( \{33^{30}\} \), and \( \{9^{5}\} \) IE \( \{31^{12}\} \),

FN(4) = “Compute”, and FN(21) = “Avg”,

\[ S_c = S_c \cup TF_{\text{Avg}} \]
\[ TF_{\text{Compute}} = \{9^{5}, 10^{5}, 11^{6}, 12^{7}, 12^{8}, 12^{9}, 12^{10}, 14^{11}\} \]
\[ TF_{\text{Avg}} = \{23^{21}, 24^{22}\} \]
\[ S_c = \{28^{1}, 29^{2}, 30^{3}, 9^{4}, 10^{5}, 11^{6}, 12^{7}, 12^{8}, 12^{9}, 12^{10}, 14^{11}, 31^{12}, 23^{21}, 24^{22}, 33^{30}\} \]

since \( \{16^{23}\} \) IE \( \{24^{25}\} \),

FN(23) = “Sum”,

\[ S_c = S_c \cup TF_{\text{Sum}} \]
\[ TF_{\text{Sum}} = \{16^{23}, 17^{24}, 18^{25}, 19^{26}, 18^{27}, 18^{28}, 20^{29}\} \]

Finally, we get \( S[0] = S[0] \cup S_c \)


Step 2: Check for more Called-to-Calculated functions since FN(30) = “main” then no more calling functions and break.

Step 3: Add scope of influence

\( \text{Slice}[1] = \text{Add}_{-}\text{Scope}_{-}\text{of}_{-}\text{Influence}(S[0]) \)

Let \( F^0 = S_0 = S[0] \)

\[ F^0 = \{9, 10, 11, 12, 14, 16, 17, 18, 20, 23, 24, 28, 29, 30, 31, 33\} \]
\[ S^0 = \{9, 10, 11, 12, 14, 16, 17, 18, 20, 23, 24, 28, 29, 30, 31, 33\} \]

\[ F^1 = \{1, 3, 5, 6, 21, 25, 34\} \]
\[ S^1 = \{1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 23, 24, 25, 28, 29, 30, 31, 33, 34\} \]

\[ F^2 = \{26\} \]
\[ S^2 = \{1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 23, 24, 25, 26, 28, 29, 30, 31, 33, 34\} \]

\[ F^3 = \{\} \]
\[ S^3 = \{1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 23, 24, 25, 26, 28, 29, 30, 31, 33, 34\} \]

And finally, the dynamic slice is shown in Figure 15.

**Example 2.** Consider trajectory T in Figure 9. Using the criterion \( C = (x, 32^{20}, \{\text{Sum}\}) \), we have \( x = (\text{Max}, \text{Num}) = (3, (10.0, 20.0, 15.0, 5.0)) \).

The step-by-step trace of the algorithm in Figure 5 follows.

**Step 1:**

\[ S[0] = S[0] \cup \]

Compute_Slice_in_Function_Name(C)

**Step 1.1:**

FN(C,q) = FN (20) = “main”

// therefore compute slice in function “main”

compute TF_{main} // as shown in Figure 9

compute LDF_{main} // as shown in Figure 10

compute DUF_{main} // as shown in Figure 10
compute TCF\_main // as shown in Figure 10
compute IRF\_main // as shown in Figure 10
compute LDRF\_main // as shown in Figure 10

Step 1.2:
Compute S = ComputeSlice( DUF\_main, TCF\_main,
IRF\_main, LDRF\_main, C).

Since C = (x, 32, 20, [Sum]) // given
LD(20, [Avg]) = [1], LT(32) = 28, \Gamma = 32

A^0 = \{28, 32\}, \quad S^0 = \{28, 32\},
A^1 = \{31\}, \quad S^1 = \{28, 31, 16, 32\},
A^2 = \{29, 30\}, \quad S^2 = \{28, 29, 30, 31, 32\},
A^3 = \{1\}, \quad S^3 = \{28, 29, 30, 31, 32\}.

S_c = S = \{28, 29, 30, 31, 32\}. Step 1.3: Check Calling-to-Called functions
Yes, since \{9\} IE \{31\}, and \{16\} IE \{32\},
FN(4) = “Compute”, and FN(13) = “Sum”,
S_c = S_c ∪ TF_{Compute} \cup TF_{Sum},
TF_{Compute} = \{9, 10, 11, 12, 13, 14, 15\},
TF_{Sum} = \{16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32\},
S_c = \{28, 29, 30, 31, 32\},
Finally, we get S[0] = S[0] ∪ S_c
= \{28, 29, 30, 31, 32\}.

Step 2: Check for more Called-to-Called functions
since FN(20) = “main” then no more calling function and break

Step 3: Add scope of influence
Slice[1] = Add_Scope_of_Influence(S[0])
Let F^0 = S_0 = S[0]
F^0 = \{9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32\},
S^0 = \{9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30, 31, 32, 33, 34\},
F^1 = \{1, 3, 5, 6, 21, 34\},
S^1 = \{1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 28, 29, 30, 31, 32, 34\},
F^2 = \{26\},
S^2 = \{1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 26, 28, 29, 30, 31, 32, 34\},
F^3 = \{\},
S^3 = \{1, 3, 5, 6, 9, 10, 11, 12, 14, 16, 17, 18, 20, 21, 26, 28, 29, 30, 31, 32, 34\}.

Fig. 15 A dynamic program slice computed based on variabl.
Avg in line 33 of the program in Figure 8
6. Conclusions

In this paper, we presented the definitions, the algorithms, and the approaches used to compute a program slice and a program segment after dicing. Some examples were shown as well.

In this work, a number of definitions and algorithms originally introduced by Korel and Laski [5] were modified in order to compute slices in classes, objects, arrays, pointers, references, dynamic allocation operators, function overloading, copy constructors, default arguments, operator overloading, inheritance, virtual functions, polymorphism, templates, and exception handling of a C++ program. These definitions and algorithms were used to implement a tool named C++Debug.

C++Debug was designed to allow ease and convenience on the part of the user. Using C++Debug, a user can interact directly with the computer in locating errors in a program. For convenience, the program provides menus to allow the user to select any one of the functions contained therein. Based on the results of the experimentation, C++Debug could generate a new slicing program that is of smaller size than the original source program. The new slicing program still preserves part of the program's original behavior for a specific input. In addition, C++Debug can be used as a tool like etrace under UNIX [14]. C++Debug can work on both C and C++.

7. References


