CS-TR-3594
UMIACS-TR-9613

January 1996

Extracting Reusable Functions by Program Slicing

Filippo Lanubile
Computer Science Department
University of Maryland
Institute for Advanced Computer Studies
College Park, Maryland 20742
lanubile@cs.umd.edu

Giuseppe Visaggio
Dipartimento di Informatica
University of Bari
Via Orabona 4, 70126 Bari, Italy
visaggio@seldi.uniba.it

ABSTRACT[1]

An alternative approach to developing reusable components from scratch is to recover them from existing systems. In this paper, we apply program slicing, introduced by Weiser, to the problem of extracting reusable functions from ill-structured programs. We extend the definition of program slice to a transform slice, one that includes statements which contribute directly or indirectly to transform a set of input variables into a set of output variables. Unlike conventional program slicing, these statements do not include neither the statements necessary to get input data nor the statements which test the binding conditions of the function. Transform slicing presupposes the knowledge that a function is performed in the code and its partial specification, only in terms of input and output data. Using domain knowledge we discuss how to formulate expectations of the functions implemented in the code. In addition to the input/output parameters of the function, the slicing criterion depends on an initial statement which is difficult to obtain for large programs. Using the notions of decomposition slice and concept validation we demonstrate how to produce a set of candidate functions, which are independent of line numbers but must be evaluated with respect to the expected behavior. Although human interaction is required, the limited size of candidate functions makes this task easier than looking for the last function instruction in the original source code.

[1] This work is supported in part by the National Science Foundation under grant 01-5-24845 and the Italian M.U.R.S.T. under the 40% project "V&V in software engineering".

1. Introduction

Although reusability is widely accepted as the key for improving productivity and quality, in the software field real practice is still far behind other engineering disciplines. One of the obstacles to a massive application of software reuse in industrial environments is that the initial building of reusable software is more costly. An experiment conducted in the Software Engineering Laboratory over a 6-year period, comparing Fortran and Ada projects [5], has shown that creating reusable software components requires a huge initial investment which is not rapidly amortized. This explains the reluctance of companies to adopt software reuse as an established practice in developing software.

An alternative approach to developing new reusable components is to recover them from existing software systems. There is great potential in this last approach because billion of lines of code have already been written by programmers. Software managers do not expect the past knowledge and experience embodied in their software portfolio to be thrown away.

Although informal software scavenging is a popular practice among programmers, it is performed using informal abstractions which exist only in the memory of the developers [29]. To be feasible on a large-scale, the code scavenging approach should be supported by automatic tools based on formal models of extraction.

Any approach related to software reuse involves some form of abstraction for software artifacts. Extracting reusable components from existing software systems means locating in the code those parts which implement the data or functional abstractions. Our work focuses on locating functional abstractions, but data abstractions can be produced through the aggregation of data structures and recovered functional components around more general abstract data types.

New programs, when well designed, have functional abstractions represented by subprograms. However, many legacy programs [7] have an inadequate design or one which has been corrupted by enhancements and patches introduced during their operational life. The result is that old programs suffer from interleaving, which expresses the merging of two or more distinct plans within some contiguous textual area of the program [41]. Plans, the abstract structures which model the programmer goals, are delocalized [31] and so it is difficult to recognize, maintain, and reuse them in other contexts. To extract reusable functions from ill-structured programs we need a decomposition method which is able to group generally non sequential sets of statements.

Program slicing is a family of program decomposition techniques based on selecting statements relevant to a computation, even if they are scattered throughout the program. Program slicing, as originally defined by Weiser [43], is based on static dataflow analysis on the flow graph of the program. A program slice can also be found in linear time as the transitive closure of a dependency graph [20]. Program slicing has been applied in program debugging, parallel processing, program testing, program integration, program understanding and software maintenance, both using the basic definition and developing variants, including program dicing [36], dynamic slicing [3, 27], decomposition slicing [22], relevant slicing [4], interface slicing [6], conditioned slicing [12], and variable slicing [25]. Conventional program slicing has been also advocated for the purpose of software reuse [6, 33]. However, program slices are often imprecise as reusable functions because they contain unnecessary statements for the function to recover. Hence this proposal of a new slicing approach, called transform slicing, which is more effective in extracting functional components from old programs.

Transform slicing presupposes the knowledge that a function is performed in a system and its partial specification, only in terms of input and output data. The aim is to take only those statements which yield the output data, both directly and indirectly, starting from the given input data. These statements, unlike conventional program slicing, do not include neither the statements necessary to get input data nor the branch and loop conditions which are used to control the activation of the function. In addition to the input/output parameters of the function, the slicing criterion depends on an initial statement. This statement, which is usually the last instruction of the function to be recovered, is difficult to identify because requires reading a lot of code. We overcome this problem by providing a scavenging algorithm which invokes transform slicing but does not depend on statement numbers. A set of candidate functions are produced and evaluated with respect to their expected behavior. Although this concept validation step is not automatic, the limited size of candidate functions makes this task easier than looking for the last function instruction in the original source code.

Since legacy systems do not always have accurate or up-to-date documentation, the application of transform slicing to the creation of reusable assets is part of a reverse engineering process, which has been designed mainly for data-strong applications, and uses information from data model representation to drive the recovery of functional components. The data model allows the expected functions to be specified in terms of their input and output data. Once these parameters have been mapped onto variables in the source code, slicing criteria are formulated and transform slicing extracts a set of cohesive functions, which implement conceptually simple tasks.

This paper is a revised and extended version of [30] and takes advantage of lessons learned from previous applications of the function extraction to legacy systems [1, 17, 21]. However, transform slicing definitions are language-independent and could also be applied using a different process model.

The rest of the paper is organized as follows. Section 2 defines some necessary terminology and introduces transform slicing at the procedural level. Section 3 extends both basic definitions and transform slicing for dealing with procedural programs. In Section 4, we describe how to elicit the specifications of the functional abstractions to be searched for and how transform slicing can be realistically applied to legacy systems for producing a set of cohesive reusable functions. In Section 5, related work on component extraction is surveyed and compared to our approach. Finally, Section 6 presents a summary and discusses possible future research directions.

2. Intraprocedural Extraction Criteria

This section deals with data flow equations applied to a program procedure. The definitions are language-independent and include unstructured programs too. In the following subsections we give basic definitions and our equations for extracting transform slices.

2.1 Background

The definitions below are used to establish a common terminology to be used in the data flow equations. We present control flow graphs and def/use graphs, as defined in [38], but we take only those dominance relations which are useful for the extraction criteria. Weiser's equations for program slicing are also presented to emphasize the differences from our slicing equations.

Here, program slicing is defined in terms of def/use graphs but some definitions appear different in style as respect to [43].

Definition 1: A *digraph G* is a pair (N, E), where N is a finite, nonempty set, and E is a subset of $N \times N - \{ (n, n) \mid n \in N \}$. The elements of N are called *nodes* and the elements of E are called *edges*. Given an edge $(n_i, n_j) \in E$, n_i is said to be *predecessor* of n_j, and n_j is said to be *successor* of n_i. *PRED*(n) and *SUCC*(n) are the set of the *predecessors* and the set of the *successors* of a node n respectively. The *indegree* of a node n, denoted *in*(n), is the number of predecessors of n, while the *outdegree* of n, denoted *out*(n), is the number of successors of n. A *walk W* in a digraph G is a sequence of nodes $n_1 n_2 \cdots n_k$ such that $k \geq 0$ and (n_i, n_{i+1}) for $i = 1, 2, \cdots, k - 1$, where k is the length of W. If W is nonempty (the length is zero) then it is called a $n_1 - n_k$ *walk*.

Definition 2: A *hammock graph G* is a digraph with two distinguished nodes: the *initial node* n_I and the *final node* n_F, satisfying the following conditions: (1) $in(n_I) = 0$ and $out(n_F) = 0$; (2) each node $n \in G$ occurs in a $n_I - n_F$ walk.

Definition 3: Let G be a hammock graph, and m and n two nodes in G, *m forward dominates n* iff every $n - n_F$ walk in G contains m; *m properly forward dominates n* iff $m \neq n$ and *m forward dominates n*; *m is the immediate forward dominator of n* iff m is the first node which properly forward dominates n on every $n - n_F$ walk. The set of forward dominators of a node n is denoted *FD*(n), the set of properly forward dominators *PFD*(n), while the immediate forward dominator is *ifd*(n).

Definition 4: A *control flow graph G* is a hammock graph which is interpreted as a program procedure. We use the term procedure to include also the main program and program functions. In the latter case the procedure has an extra output parameter corresponding to the value returned by the function.

The nodes of a control flow graph represent elementary program statements such as assignments, input/output instructions, branch and loop conditions, unconditional branches, and procedure calls. The initial and the final nodes represent the entry and exit points of the procedure respectively. The edges represent control flow transfers between statements. We give a wider interpretation than in [38], because we represent unconditional GOTOs as control flow graph nodes to deal with unstructured programs. We can also represent programs with multiple entry and exit points. In this case there will be two kinds of special nodes: start nodes N_S and *halt nodes* N_H to represent the multiple entry points and exit points, respectively. These nodes must satisfy the following conditions:

(i) for each $s \in N_S : PRED(s) = \{n_I\}$

(ii) for each $h \in N_H : SUCC(h) = \{n_F\}$

A control flow graph still has a ingle initial node which is connected to start nodes by edges of the form (n_I, s). Analogously the unique final node is linked to halt nodes by (h, n_F) edges.

Definition 5: Let G be a control flow graph and n a node in G. A statement m is *conditioned* by n iff m occurs in a $n-ifd(n)$ walk, excluding the endpoints n and $ifd(n)$. The set of statements conditioned by n is denoted $INFL(n)$. From this definition we can infer that $INFL(n)$ is empty iff $out(n) \leq 1$. Then n for a nonempty $INFL(n)$ represents a condition branch or a condition loop statement.

Definition 6: A *def/use graph* is a quadruple $\boldsymbol{G} = (G, \Sigma, D, U)$, where G is the control flow graph representing a program procedure, Σ is a finite set of symbols naming variables in the program procedure, $D : N_G \rightarrow \mathbb{P}(\Sigma)$, and $U : N_G \rightarrow \mathbb{P}(\Sigma)$ are functions mapping the nodes of G in the set of variables which are defined or used in the statements corresponding to nodes.

A variable x is *defined* in a statement s if an execution of s assigns a value to x, while a variable x is *used* in a statement s if an execution of s requires the value of x to be evaluated. Assignment statements have defined variables in the left-part and used variables in the right-part; input statements have only defined variables while output statements have only used variables; variables in branch and loop conditions are only used, while unconditional branches have neither used nor defined variables.

Example 1: Let us consider a program, already appeared as example in [43]. The def/use graph is shown in Figure 1. The numbers of the initial and final nodes represent, respectively, the initial and final program statements, while the other nodes are numbered according with the positions of the executable statements.

```
1.    begin
2.      read ( x, y );
3.      total := 0.0;
4.      sum := 0.0;
5.      if x <= 1
6.           then sum := y
7.           else    begin
8.                         read (z);
9.                         total := x * y;
10.                   end;
11.     write (total, sum);
12.   end.
```

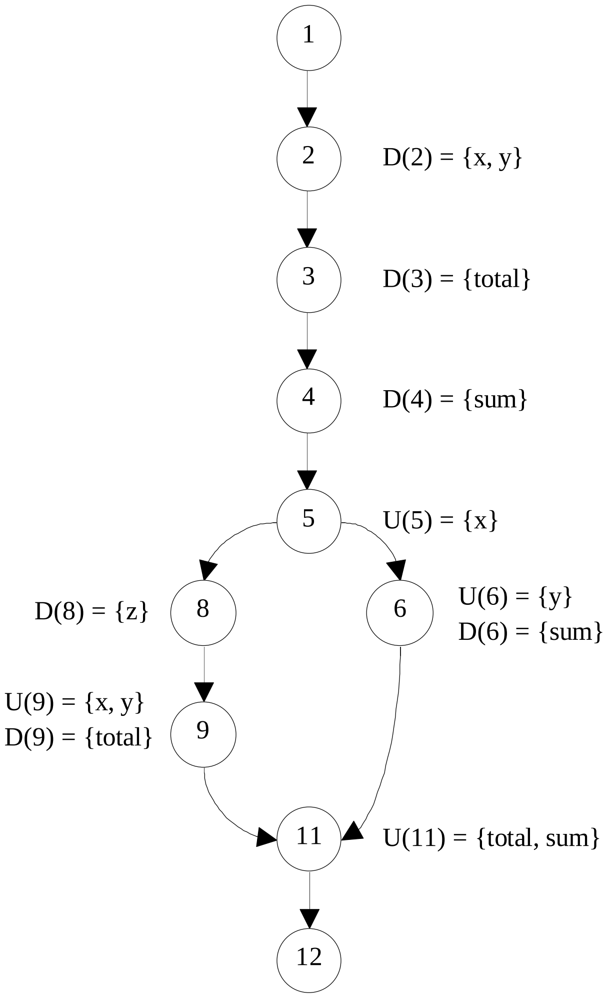

Fig. 1. Def/Use graph for examples 1, 2, 3

The computational model we adopt includes only scalar variables but can be extended to include other constructs such as structures, arrays and pointers. A structure variable or “record” in other programming languages can be seen as the union of its component variables. Thus, defining or using a structure variable implies that all its component variables are defined or used, respectively. In the same way, defining or using a component variable implies that the including structure variable is defined or used, respectively. A safe but conservative approach in static data flow analysis treats all the elements of an array or specified by a pointer as a single object. More refined approaches valid for arrays and pointers in C programs are defined in [24, 37].

In the definitions below we assume we have a def/use graph $\boldsymbol{G} = (G, \Sigma, D, U)$ and a program procedure P represented by $\boldsymbol{G}$.

Definition 7: A *slicing criterion* is a pair $C = \langle\, i, V \,\rangle$, where $i \in N_G$ and $\mathrm{V} \subseteq \Sigma$. In the program procedure P, a slicing criterion is made up of one statement and a subset of variables.

Definition 8: A *slice S* on a slicing criterion $C = \langle\, i, V \,\rangle$, denoted S_C, is an executable subset of P containing all the statements which contribute to the values of V just before statement i is executed.

Definition 9: Let $C = \langle\, i, V \,\rangle$ be a slicing criterion. The set of variables relevant to C, when program execution is at statement n, denoted $R_C{}^0(n)$, is defined as follows:

$$\begin{aligned} R_C{}^0(n) = {} & \{\, v \in V \mid n = i \,\} \cup \\ & \{\, U(n) \mid D(n) \cap R_C{}^0(SUCC(n)) \neq \varnothing \,\} \cup \\ & \{\, R_C{}^0(SUCC(n)) - D(n) \,\} \end{aligned}$$

$R_C{}^0(n)$ includes the variables which have potential effects on the def-use chain ending in V. Search starts from node i and goes backward. The first subset expresses the base case. The second dictates that variables which are used to assign values to other variables, already marked as relevant, become relevant. The third case excludes a relevant variable when it has been modified.

Definition 10: Let $C = \langle\, i, V \,\rangle$ be a slicing criterion. The set of statements relevant to C, denoted $S_C{}^0$, is defined as follows:

$$S_C{}^0 = \{\, n \in G \mid D(n) \cap R_C{}^0(SUCC(n)) \neq \varnothing \,\}$$

$S_C{}^0$ includes the statements whose execution can directly influence the values of relevant variables.

Definition 11: Let $C = \langle\, i, V \,\rangle$ be a slicing criterion. The set of conditional statements which control the execution of the statements in S_C^0, denoted B_C^0, is defined as follows:

$$B_C^0 = \{\, b \in G \mid INFL(b) \cap S_C^0 \neq \emptyset \,\}$$

In the following, the building of S_C is defined recursively on the set of variables and statements which have either direct or indirect influence on V. Starting from zero, the superscripts represent the level of recursion.

$$R_C^{i+1}(n) = R_C^{i}(n) \bigcup_{b \in B_C^{i}} R_{\langle b,\, U(b)\rangle}^{0}(n) \tag{1}$$

$$S_C^{i+1} = \{\, n \in G \mid D(n) \cap R_C^{i+1}(SUCC\,(n)) \neq \emptyset \,\} \cup B_C^{i} \tag{2}$$

$$B_C^{i+1} = \{\, b \in G \mid INFL(b) \cap S_C^{i+1} \neq \emptyset \,\} \tag{3}$$

The full definition includes the conditional statements with an indirect influence on a slice, the control variables which are evaluated in the logical expression, and the statements which influence the control variables. The iteration continues until no new variables are relevant and so no new statements may be included. In other words $\mathrm{S}_C = \mathrm{S}_C^{f+1}$ where f is an iteration step such that $\forall\ n \in N : R_C^{f+1}(n) = R_C^{f}(n) = R_C(n)$.

Example 2: Let us consider the program in the Example 1. Given the slicing criterion $C = \langle$ 11, $\{$ total$\}\rangle$, for each executable statement we have the following sets:

$R_C^0(11) = \{\text{total}\}$ $R_C^0(9) = \{\text{x, y}\}$ $R_C^0(8) = \{\text{x, y}\}$

$R_C^0(6) = \{\text{total}\}$ $R_C^0(5) = \{\text{total, x, y}\}$ $R_C^0(4) = \{\text{total, x, y}\}$

$R_C^0(3) = \{\text{x, y}\}$ $R_C^0(2) = \emptyset$

$S_C^0 = \{2, 3, 9\}$ $B_C^0 = \{5\}$

$R_{\langle 5,\, \{\text{x}\}\rangle}^0(5) = \{\text{x}\}$ $R_{\langle 5,\, \{\text{x}\}\rangle}^0(4) = \{\text{x}\}$ $R_{\langle 5,\, \{\text{x}\}\rangle}^0(3) = \{\text{x}\}$ $R_{\langle 5,\, \{\text{x}\}\rangle}^0(2) = \emptyset$

$R_C^1(11) = \{\text{total}\}$ $R_C^1(9) = \{\text{x, y}\}$ $R_C^1(8) = \{\text{x, y}\}$

$R_C^1(6) = \{\text{total}\}$ $R_C^1(5) = \{\text{total, x, y}\}$ $R_C^1(4) = \{\text{total, x, y}\}$

$R_C^1(3) = \{\text{x, y}\}$ $R_C^1(2) = \emptyset$

$S_C = S_C^1 = \{2, 3, 5, 9\}$

Unconditional branches cannot be caught by these definitions because the statements have no defined or used variables. However, their omission can bias the behavior of the slice resulting in an incorrect projection of the program. Although restricted to C language, in [24] an algorithm is presented to collect *goto* statements and a set of rules are given to pick up *break* and *continue* statements. These algorithms can easily be extended to other languages for dealing with other kinds of branches. Although we use them when slicing, the scope of this paper does not include slicing extensions for unconditional branches.

2.2 Transform Slice

Let $\boldsymbol{G} = (G, \Sigma, D, U)$ be a def/use graph, and P a program procedure represented by $\boldsymbol{G}$. The following definitions are given to extract the implementation of functional abstractions.

Definition 12: A *transform slicing criterion* is a triple $C = \langle i, V_{\mathrm{inp}}, V_{\mathrm{out}} \rangle$, where $i \in N_G$ and V_{inp}, V_{out} are both subsets of Σ.

Definition 13: A *transform slice* on a transform slicing criterion $C = \langle i, V_{inp}, V_{out} \rangle$, denoted TrS_C , is an executable subset of P containing all the statements which contribute either directly or indirectly to the values of V_{out} starting from the values of V_{inp}, just before statement i is executed.

Definition 14: Let $C = \langle i, V_{inp}, V_{out} \rangle$ be a transform slicing criterion. The set of variables relevant to C, when program execution is at statement n, denoted $TrR_C{}^0(n)$, is defined as follows:

$$\begin{aligned} TrR_C{}^0(n) = {} & \{ v \in V_{out} \mid n = i \} \cup \\ & \{ U(n) - V_{inp} \mid D(n) \cap TrR_C{}^0(SUCC(n)) \neq \emptyset \} \cup \\ & \{ TrR_C{}^0(SUCC(n)) - D(n) \} \end{aligned}$$

$TrR_C{}^0(n)$ includes the variables which have potential effects on V_{out} , with the exclusion of variables coming before V_{inp} in the use-definition chain. Like in definition 9, the search starts from node i and goes backward, but this time it stops when the variables in V_{inp} have been found (second subset in the definition).

Definition 15: Let $C = \langle i, V_{inp}, V_{out} \rangle$ be a transform slicing criterion. The set of statements relevant to C, denoted $TrS_C{}^0$, is defined as follows:

$$TrS_C{}^0 = \{ n \in G \mid D(n) \cap TrR_C{}^0(SUCC(n)) \neq \emptyset \}$$

This definition is substantially equal to definition 10.

Definition 16: Let $C = \langle i, V_{inp}, V_{out} \rangle$ be a transform slicing criterion. The set of conditional statements which control the execution of the statements in $TrS_C{}^0$, denoted $TrB_C{}^0$, is defined as follows:

$$TrB_C{}^0 = \{ b \in G \mid INFL(b) \cap TrS_C{}^0 \neq \emptyset \text{ and } i \notin INFL(b) \}$$

$TrB_C{}^0$ restricts definition 11 because it includes the conditional statements which influence the statements in $TrS_C{}^0$, only if they do not condition the statement *i* too. In fact, a conditional statement influencing the slicing criterion statement *i* means that the overall execution of the sliced component could be excluded as a result of the evaluation of the condition. Thus, the conditional instruction should remain outside as part of the program manager which invokes the sliced component. There are three main cases where the exclusion of a conditional statement is useful when isolating a functional component:

a) A program procedure performs multiple different functions which are activated by a function tag

b) A program procedure contains the pre-conditions for the function

c) A program procedure performs the function iteratively

As an effect of definition 16, branch conditions (cases *a* and *b*) and loop conditions (case *c*) will not be part of the transform slice.

Like Weiser's slice, the transform slice is built recursively. Starting from zero, the superscripts represent the level of recursion.

$$TrR_C{}^{i+1}(n) = TrR_C{}^{i}(n) \bigcup_{b \in TrB_C{}^{i}} TrR_{\langle b,\, V_{inp},\, U(b) - V_{inp} \rangle}{}^{0}(n) \qquad (4)$$

$$TrS_C{}^{i+1} = \{ n \in G \mid D(n) \cap TrR_C{}^{i+1}(SUCC\,(n)) \neq \emptyset \} \cup TrB_C{}^{i} \qquad (5)$$

$$TrB_C{}^{i+1} = \{ b \in G \mid INFL(b) \cap TrS_C{}^{i+1} \neq \emptyset \text{ and } i \notin INFL(b) \} \qquad (6)$$

The iteration is similar to that for conventional slicing, with the exception that equation (4) excludes input variables from becoming output variables when slicing starts from conditional statements, and equation (6) is modified according to definition 16. The rule for stopping iteration remains unchanged. Input/output statements which deal with variables of the transform slicing criterion are not included because in our definition the transform slice is input-restricted as regards variables in the transform slicing criterion and output-restricted because all the output statements are removed.

The transform slice is also a def/use graph which can be packaged as a distinct module. A complement of the direct slice may be derived, working as a caller which activates the transform module. However, the complement computation will not be shown here, because is beyond the scope of this paper.

Example 3: Let us consider the program example in the Example 1. Given the transform slicing criterion $C = \langle 11, \{x, y\}, \{total\}\rangle$, for each executable statement we have the following sets:

$TrR_C{}^0(11) = \{total\}$ $TrR_C{}^0(9) = \emptyset$ $TrR_C{}^0(8) = \emptyset$

$TrR_C{}^0(6) = \{total\}$ $TrR_C{}^0(5) = \{total\}$ $TrR_C{}^0(4) = \{total\}$

$TrR_C{}^0(3) = \emptyset$ $TrR_C{}^0(2) = \emptyset$

$TrS_C{}^0 = \{3, 9\}$ $TrB_C{}^0 = \{5\}$

$TrR_{\langle 5, \{x, y\}, \emptyset\rangle}{}^0(5) = \emptyset$ $TrR_{\langle 5, \{x, y\}, \emptyset\rangle}{}^0(4) = \emptyset$

$TrR_{\langle 5, \{x, y\}, \emptyset\rangle}{}^0(3) = \emptyset$ $TrR_{\langle 5, \{x, y\}, \emptyset\rangle}{}^0(2) = \emptyset$

$TrR_C{}^1(11) = \{total\}$ $TrR_C{}^1(9) = \emptyset$ $TrR_C{}^1(8) = \emptyset$

$TrR_C{}^1(6) = \{total\}$ $TrR_C{}^1(5) = \{total\}$ $TrR_C{}^1(4) = \{total\}$

$TrR_C{}^1(3) = \emptyset$ $TrR_C{}^1(2) = \emptyset$

$TrS_C = TrS_C{}^1 = \{3, 5, 9\}$

In this example, the only effect of applying transform slicing with respect to conventional slicing was that the input statement *2* was excluded because the two variables *x* and *y* have been declared as input variables in the transform slicing criterion and hence they are considered as input parameters to the extracted function. On the contrary, the conditional statement *5* have been included because it does not control the initial statement *11* in the transform slicing criterion and so it is considered part of the function to be recovered.

After having considered this base example, we will show other new examples corresponding to the three cases for which the exclusion of a conditional statement is advocated.

Example 4 (case *a*): Let us consider a program which computes the sum and product of first n numbers, using a single loop. The def/use graph is shown in Figure 2. Nodes are numbered as in the first example.

```
1.    begin
2.      read ( n );
3.      if n > 0
4.          then   begin
5.                      i := 1;
6.                      sum := 0;
7.                      prod := 1;
8.                      while i <= n do
9.                          begin
10.                             read ( k );
11.                             sum := sum +k;;
12.                             prod := prod * k;
13.                             i := i + 1;
14.                         end;
15.                     write ( sum );
16.                     write ( prod );
17.                end;
18.   end.
```

Given the slicing criterion $C = \langle 15, \{\text{sum}\}\rangle$, for each executable statement we have the following sets:

$R_C{}^0(15) = \{\text{sum}\}$ $R_C{}^0(13) = \{\text{sum}\}$ $R_C{}^0(12) = \{\text{sum}\}$

$R_C{}^0(11) = \{\text{k, sum}\}$ $R_C{}^0(10) = \{\text{sum}\}$ $R_C{}^0(8) = \{\text{sum}\}$

$R_C{}^0(7) = \{\text{sum}\}$ $R_C{}^0(6) = \varnothing$ $R_C{}^0(5) = \varnothing$

$R_C{}^0(3) = \varnothing$ $R_C{}^0(2) = \varnothing$

$S_C{}^0 = \{6, 10, 11\}$ $B_C{}^0 = \{3, 8\}$

$R_{\langle 8, \{\text{i, n}\}\rangle}{}^0(8) = \{\text{i, n}\}$ $R_{\langle 8, \{\text{i, n}\}\rangle}{}^0(13) = \{\text{i, n}\}$ $R_{\langle 8, \{\text{i, n}\}\rangle}{}^0(12) = \{\text{i, n}\}$

$R_{\langle 8, \{\text{i, n}\}\rangle}{}^0(11) = \{\text{i, n}\}$ $R_{\langle 8, \{\text{i, n}\}\rangle}{}^0(10) = \{\text{i, n}\}$ $R_{\langle 8, \{\text{i, n}\}\rangle}{}^0(7) = \{\text{i, n}\}$

$R_{\langle 8, \{\text{i, n}\}\rangle}{}^0(6) = \{\text{i, n}\}$ $R_{\langle 8, \{\text{i, n}\}\rangle}{}^0(5) = \{\text{n}\}$ $R_{\langle 8, \{\text{i, n}\}\rangle}{}^0(3) = \{\text{n}\}$

$R_{\langle 8, \{\text{i, n}\}\rangle}{}^0(2) = \varnothing$

$R_{\langle 3, \{\text{n}\}\rangle}{}^0(3) = \{\text{n}\}$ $R_{\langle 3, \{\text{n}\}\rangle}{}^0(2) = \varnothing$

$R_C{}^1(15) = \{\text{sum}\}$ $R_C{}^1(13) = \{\text{i, n, sum}\}$ $R_C{}^1(12) = \{\text{i, n, sum}\}$

$R_C{}^1(11) = \{\text{i, k, n, sum}\}$ $R_C{}^1(10) = \{\text{i, n, sum}\}$ $R_C{}^1(8) = \{\text{i, n, sum}\}$

$R_C{}^1(7) = \{\text{i, n, sum}\}$ $R_C{}^1(6) = \{\text{i, n}\}$ $R_C{}^1(5) = \{\text{n}\}$

$R_C{}^1(3) = \{\text{n}\}$ $R_C{}^1(2) = \varnothing$

$S_C{}^1 = \{2, 3, 5, 6, 8, 10, 11, 13\}$ $B_C{}^1 = \{3, 8\}$

Shortly, $\forall\ n \in N : R_C{}^2(n) = R_C{}^1(n)$

$S_C = S_C{}^2 = \{2, 3, 5, 6, 8, 10, 11, 13\}$

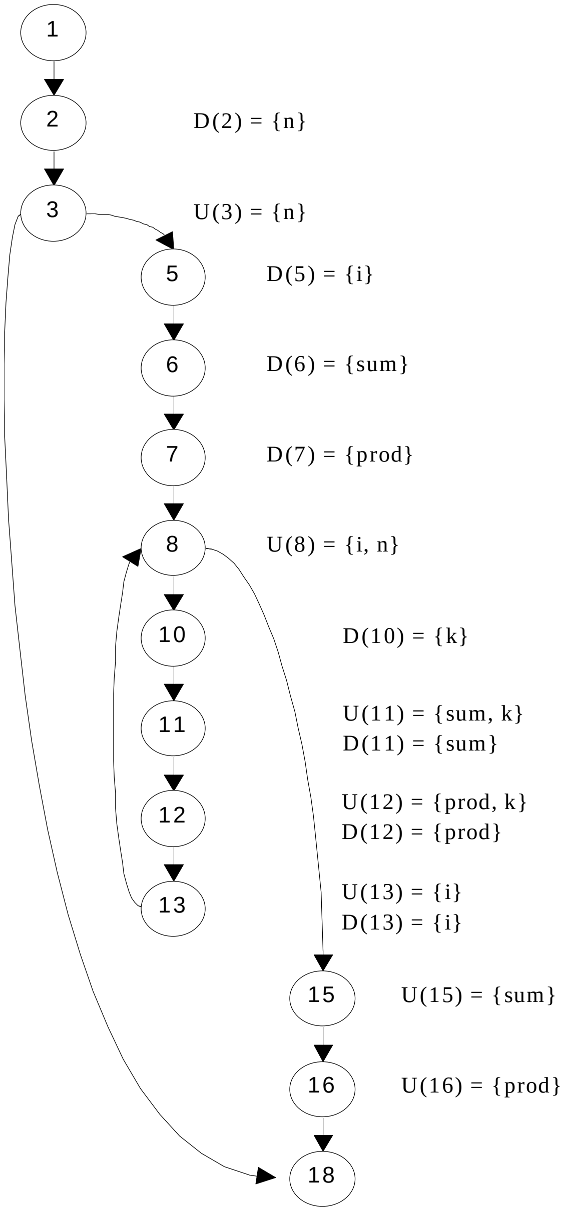

Fig. 2. Def/Use graph for example 4

Let us consider now, the results from applying transform slicing. The summation function can be modeled as $sum = \mathrm{f}\,(n)$ and so the transform slicing criterion is $C = \langle 15, \{\mathrm{n}\}, \{\mathrm{sum}\}\rangle$. For each executable statement we have the following sets:

$TrR_C^0(15) = \{\mathrm{sum}\}$ $TrR_C^0(13) = \{\mathrm{sum}\}$ $TrR_C^0(12) = \{\mathrm{sum}\}$
$TrR_C^0(11) = \{\mathrm{k, sum}\}$ $TrR_C^0(10) = \{\mathrm{sum}\}$ $TrR_C^0(8) = \{\mathrm{sum}\}$
$TrR_C^0(7) = \{\mathrm{sum}\}$ $TrR_C^0(6) = \emptyset$ $TrR_C^0(5) = \emptyset$
$TrR_C^0(3) = \emptyset$ $TrR_C^0(2) = \emptyset$
$TrS_C^0 = \{6, 10, 11\}$ $TrB_C^0 = \{8\}$
$TrR_{\langle 8, \{\mathrm{i}\}\rangle}^0(8) = \{\mathrm{i}\}$ $TrR_{\langle 8, \{\mathrm{i}\}\rangle}^0(13) = \{\mathrm{i}\}$ $TrR_{\langle 8, \{\mathrm{i}\}\rangle}^0(12) = \{\mathrm{i}\}$
$TrR_{\langle 8, \{\mathrm{i}\}\rangle}^0(11) = \{\mathrm{i}\}$ $TrR_{\langle 8, \{\mathrm{i}\}\rangle}^0(10) = \{\mathrm{i}\}$ $TrR_{\langle 8, \{\mathrm{i}\}\rangle}^0(7) = \{\mathrm{i}\}$
$TrR_{\langle 8, \{\mathrm{i}\}\rangle}^0(6) = \{\mathrm{i}\}$ $TrR_{\langle 8, \{\mathrm{i}\}\rangle}^0(5) = \emptyset$ $TrR_{\langle 8, \{\mathrm{i}\}\rangle}^0(3) = \emptyset$
$TrR_{\langle 8, \{\mathrm{i}\}\rangle}^0(2) = \emptyset$
$TrR_C^1(15) = \{\mathrm{sum}\}$ $TrR_C^1(13) = \{\mathrm{i, sum}\}$ $TrR_C^1(12) = \{\mathrm{i, sum}\}$
$TrR_C^1(11) = \{\mathrm{i, k, sum}\}$ $TrR_C^1(10) = \{\mathrm{i, sum}\}$ $TrR_C^1(8) = \{\mathrm{i, sum}\}$
$TrR_C^1(7) = \{\mathrm{i, sum}\}$ $TrR_C^1(6) = \{\mathrm{i}\}$ $TrR_C^1(5) = \emptyset$
$TrR_C^1(3) = \emptyset$ $TrR_C^1(2) = \emptyset$
$TrS_C^1 = \{5, 6, 8, 10, 11, 13\}$ $TrB_C^1 = \{8\}$
Shortly, $\forall\ n \in N : TrR_C^2(n) = TrR_C^1(n)$
$TrS_C = TrS_C^2 = \{5, 6, 8, 10, 11, 13\}$

With respect to the conventional slice, the transform slice does not include statement 2 which reads the input variable of the function, and statement 3 which contains the predicate which implements the precondition of the summation function.

Example 5 (case b): Let us consider a program which computes the sum or product of first n numbers, according to the value of a flag. There is no control for the preconditions of the two functions. The def/use graph is shown in Figure 3. Nodes are numbered as in the previous examples.

```
1.    begin
2.      read ( n );
3.      read ( flag );
4.      i :=1;
5.      if flag = 1
6.           then   begin
7.                       sum := 0;
8.                       while i <= n do
9.                            begin
10.                               read ( k );
11.                               sum := sum +k;;
12.                               i := i + 1;
13.                           end;
14.                      write ( sum );
15.                 end
16.          else   begin
17.                      prod := 1;
18.                      while i <= n do
19.                           begin
20.                               read ( k );
21.                               prod := prod * k;
22.                               i := i + 1;
23.                           end;
24.                      write ( prod );
25.                 end;
26.   end.
```

Given the slicing criterion $C = \langle 14, \{sum\}\rangle$, for each executable statement we have the following sets:

$R_C^0(14) = \{sum\}$ $R_C^0(12) = \{sum\}$ $R_C^0(11) = \{k, sum\}$

$R_C^0(10) = \{sum\}$ $R_C^0(8) = \{sum\}$ $R_C^0(7) = \emptyset$

$R_C^0(5) = \emptyset$ $R_C^0(4) = \emptyset$ $R_C^0(3) = \emptyset$ $R_C^0(2) = \emptyset$

$S_C^0 = \{7, 10, 11\}$ $B_C^0 = \{5, 8\}$

$R_{\langle 8, \{i, n\}\rangle}^0(8) = \{i, n\}$ $R_{\langle 8, \{i, n\}\rangle}^0(12) = \{i, n\}$ $R_{\langle 8, \{i, n\}\rangle}^0(11) = \{i, n\}$

$R_{\langle 8, \{i, n\}\rangle}^0(10) = \{i, n\}$ $R_{\langle 8, \{i, n\}\rangle}^0(7) = \{i, n\}$ $R_{\langle 8, \{i, n\}\rangle}^0(5) = \{i, n\}$

$R_{\langle 8, \{i, n\}\rangle}^0(4) = \{n\}$ $R_{\langle 8, \{i, n\}\rangle}^0(3) = \{n\}$ $R_{\langle 8, \{i, n\}\rangle}^0(2) = \emptyset$

$R_{\langle 5, \{flag\}\rangle}^0(5) = \{flag\}$ $R_{\langle 5, \{flag\}\rangle}^0(4) = \{flag\}$ $R_{\langle 5, \{flag\}\rangle}^0(3) = \emptyset$

$R_{\langle 5, \{flag\}\rangle}^0(2) = \emptyset$

$R_C^1(14) = \{sum\}$ $R_C^1(12) = \{i, n, sum\}$ $R_C^1(11) = \{i, k, n, sum\}$

$R_C^1(10) = \{i, n, sum\}$ $R_C^1(8) = \{i, n, sum\}$ $R_C^1(7) = \{i, n\}$

$R_C^1(5) = \{flag, i, n\}$ $R_C^1(4) = \{flag, n\}$ $R_C^1(3) = \{n\}$ $R_C^0(2) = \emptyset$

$S_C^1 = \{2, 3, 4, 5, 7, 8, 10, 11, 12\}$ $B_C^0 = \{5, 8\}$

Shortly, $\forall\ n \in N : R_C{}^2(n) = R_C{}^1(n)$

$S_C = S_C{}^2 = \{2, 3, 4, 5, 7, 8, 10, 11, 12\}$

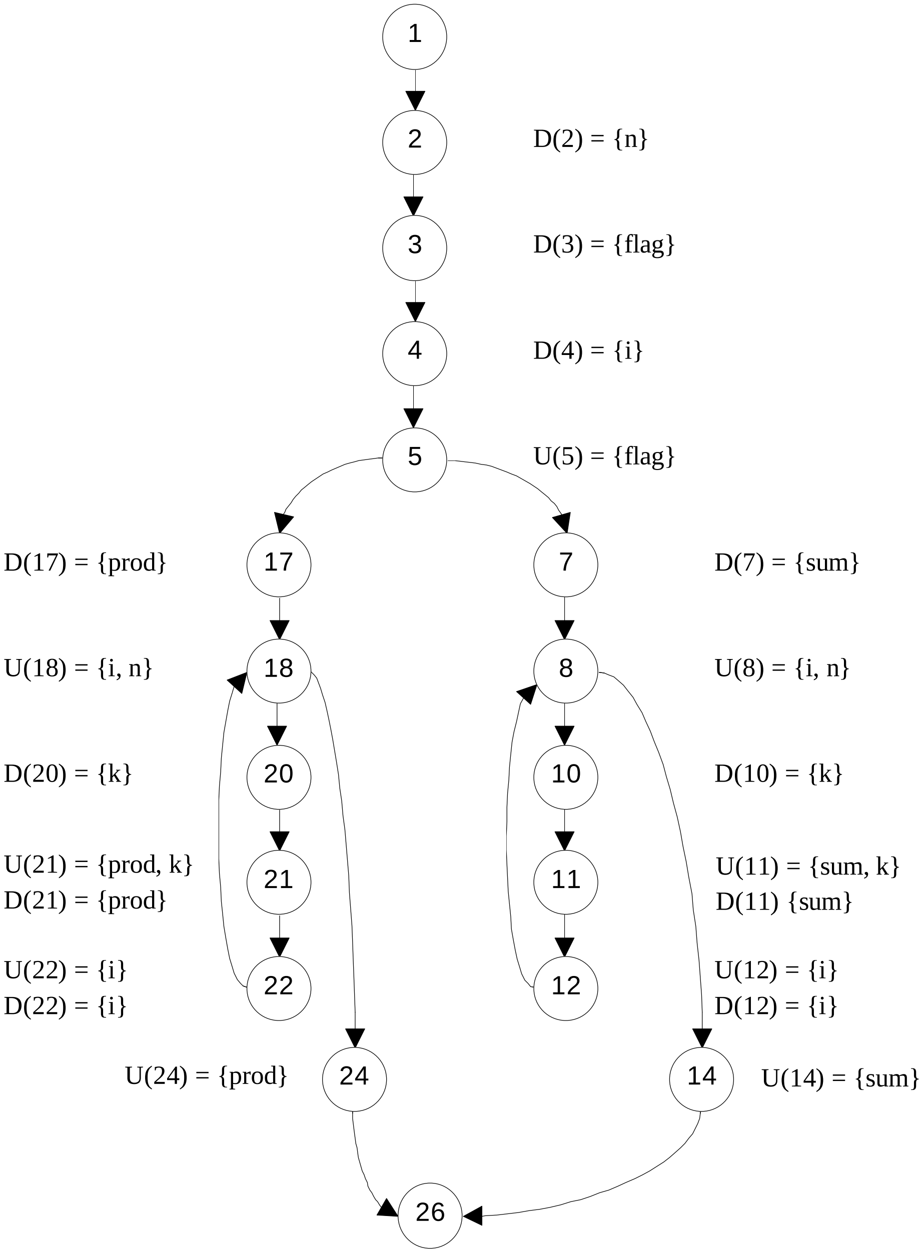

Fig. 3. Def/Use graph for example 5

Let us consider now, the results from applying transform slicing to extract the summation function. The transform slicing criterion is $C = \langle 14, \{\text{n}\}, \{\text{sum}\}\rangle$. For each executable statement we have the following sets:

$TrR_C{}^0(14) = \{\text{sum}\}$ $TrR_C{}^0(12) = \{\text{sum}\}$ $TrR_C{}^0(11) = \{\text{k, sum}\}$

$TrR_C{}^0(10) = \{\text{sum}\}$ $TrR_C{}^0(8) = \{\text{sum}\}$ $TrR_C{}^0(7) = \varnothing$

$TrR_C{}^0(5) = \varnothing$ $TrR_C{}^0(4) = \varnothing$ $TrR_C{}^0(3) = \varnothing$ $TrR_C{}^0(2) = \varnothing$

$TrS_C{}^0 = \{7, 10, 11\}$ $TrB_C{}^0 = \{8\}$

$TrR_{\langle 8, \{\text{i}\}\rangle}{}^0(8) = \{\text{i}\}$ $TrR_{\langle 8, \{\text{i}\}\rangle}{}^0(12) = \{\text{i}\}$ $TrR_{\langle 8, \{\text{i}\}\rangle}{}^0(11) = \{\text{i}\}$

$TrR_{\langle 8, \{\text{i}\}\rangle}{}^0(10) = \{\text{i}\}$ $TrR_{\langle 8, \{\text{i}\}\rangle}{}^0(7) = \{\text{i}\}$ $TrR_{\langle 8, \{\text{i}\}\rangle}{}^0(5) = \{\text{i}\}$

$TrR_{\langle 8, \{\text{i}\}\rangle}{}^0(4) = \varnothing$ $TrR_{\langle 8, \{\text{i}\}\rangle}{}^0(3) = \varnothing$ $TrR_{\langle 8, \{\text{i}\}\rangle}{}^0(2) = \varnothing$

$TrR_C{}^1(14) = \{\text{sum}\}$ $TrR_C{}^1(12) = \{\text{i, sum}\}$ $TrR_C{}^1(11) = \{\text{i, k, sum}\}$

$TrR_C{}^1(10) = \{\text{i, sum}\}$ $TrR_C{}^1(8) = \{\text{i, sum}\}$ $TrR_C{}^1(7) = \{\text{i}\}$

$TrR_C{}^1(5) = \{\text{i}\}$ $TrR_C{}^1(4) = \varnothing$ $TrR_C{}^1(3) = \varnothing$ $TrR_C{}^1(2) = \varnothing$

$TrS_C{}^1 = \{4, 7, 8, 10, 11, 12\}$ $TrB_C{}^1 = \{8\}$

Shortly, $\forall\ n \in N : TrR_C{}^2(n) = TrR_C{}^1(n)$

$TrS_C = TrS_C{}^2 = \{4, 7, 8, 10, 11, 12\}$

With respect to the conventional slice, the transform slice does not include statement 2 which reads the input variable of the function, and statements 3 and 5 which read and control, respectively, the function code which is used to dynamically choose the function to be executed.

Example 6 (case c): Let us consider a program fragment, already appeared as example in [12], which computes university taxes and room fees from the student requests of enrollment. The application is typical of batch programs, where each input record is processed inside a loop until end of file is reached. The def/use graph is shown in Figure 4. Nodes are numbered as in the previous examples.

```
    ..........
1.  read (tax_enr);
2.  read (tax_fix);
3.  read (tax_ex);
4.  read (tax_oc);
5.  read (room_max);
6.  while not eof (f_req) do
7.    begin
8.        read (f_req, stud);
9.        if stud.year = 1 then
10.           begin
11.               tax := tax_enr + tax_fix + tax_ex;
12.               room_fee := 3 * room_max / 4;
13.           end
14.       else   if stud.year >= 2 and stud.year <= 5 then
15.                   begin
16.                       tax := tax_fix + tax_oc;
17.                       room_fee := 2 * room_max / 3;
18.                   end
19.               else   begin
20.                           tax := tax_fix + tax_oc;
21.                           room_fee := room_max;
22.                       end;
23.       writeln (“Name: “, stud_name);
24.       writeln (“Mat: “, stud.mat);
25.       writeln (“Tax: “, tax);
26.       writeln (“Room fee: “, room_fee);
27.   end;
    .............
```

Let us suppose, we are interested in how the university taxes are computed. Given the slicing criterion $C = \langle\ 25,\ \{\text{tax}\}\rangle$, we have the following sets of relevant statements (for the sake of brevity, we omit the sets of variables relevant to C):

$S_C^0 = \{1, 2, 3, 4, 11, 16, 20\}$ $\qquad$ $B_C^0 = \{6, 9, 14\}$

$S_C^1 = \{1, 2, 3, 4, 6, 8, 9, 11, 14, 16, 20\}$ $\qquad$ $B_C^1 = \{6, 9, 14\}$

$S_C = S_C^2 = \{1, 2, 3, 4, 6, 8, 9, 11, 14, 16, 20\}$

Let us consider now, the results from applying transform slicing to extract the function which computes the university tax of a student. According to the alternative ways of modeling the input of the function, there will be different transform slices.

If the transform slicing criterion is $C_1 = \langle\ 25,\ \{\text{tax\_enr, tax\_fix, tax\_ex, tax\_oc, stud}\},\ \{\text{tax}\}\rangle$, we have the following sets of relevant statements:

$TrS_{C_1}^0 = \{11, 16, 20\}$ $\qquad$ $TrB_{C_1}^0 = \{9, 14\}$

$TrS_{C_1}^1 = \{9, 11, 14, 16, 20\}$ $\qquad$ $TrB_{C_1}^1 = \{9, 14\}$

$TrS_{C_1} = TrS_{C_1}^2 = \{9, 11, 14, 16, 20\}$

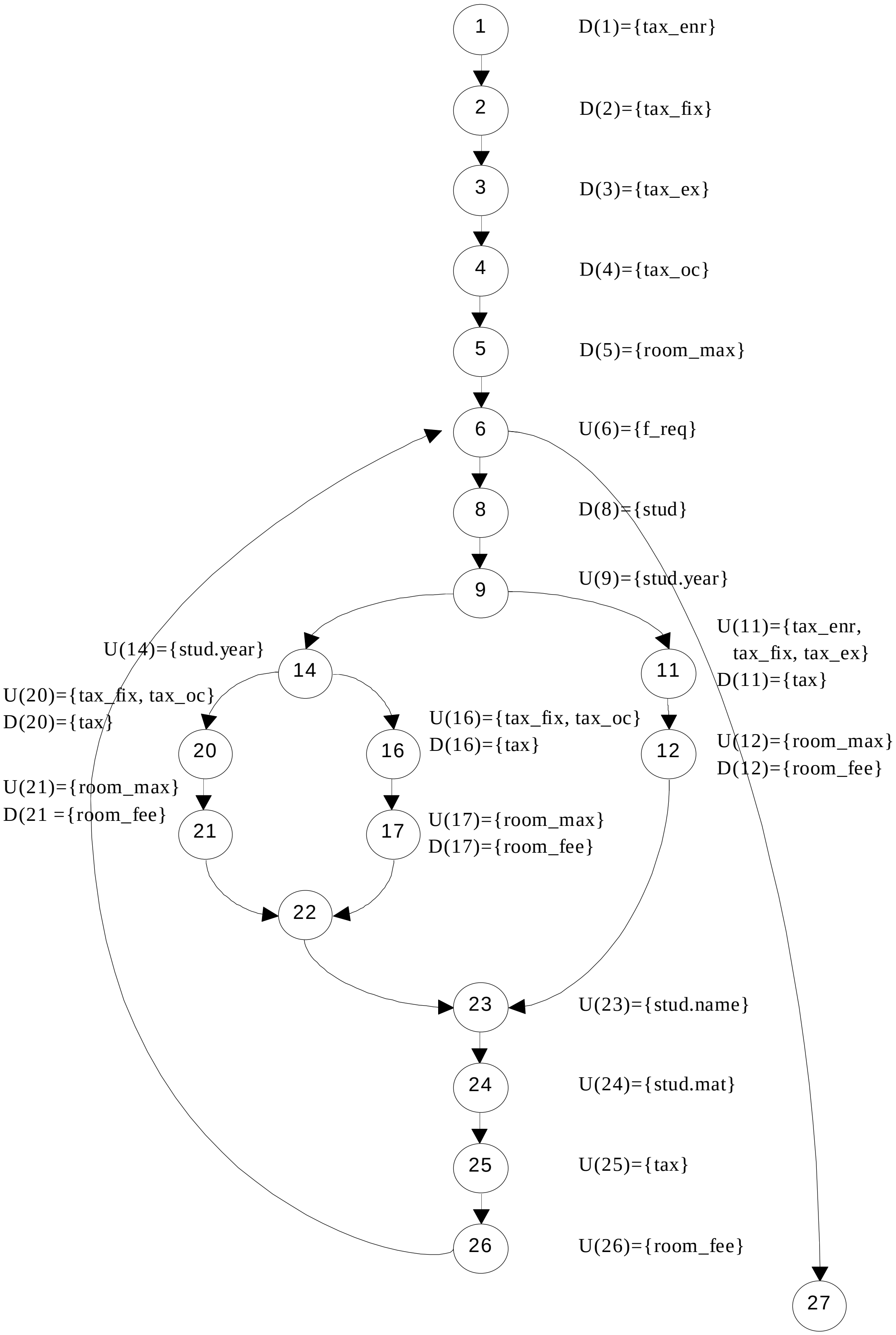

Fig. 4. Def/Use graph for example 6

If the transform slicing criterion is $C_2 = \langle\ 25,\ \{\text{tax\_enr, tax\_fix, tax\_ex, tax\_oc}\},\ \{\text{tax}\}\rangle$, we have the following sets of relevant statements:

$TrS_{C_2}^{\ 0} = \{11, 16, 20\}$ $\qquad$ $TrB_{C_2}^{\ 0} = \{9, 14\}$

$TrS_{C_2}^{\ 1} = \{8, 9, 11, 14, 16, 20\}$ $\qquad$ $TrB_{C_2}^{\ 1} = \{9, 14\}$

$TrS_{C_2} = TrS_{C_2}^{\ 2} = \{8, 9, 11, 14, 16, 20\}$

Finally, if the transform slicing criterion is $C_3 = \langle\ 25,\ \{\text{stud}\},\ \{\text{tax}\}\rangle$, we have the following sets of relevant statements:

$TrS_{C_3}^{\ 0} = \{11, 16, 20\}$ $\qquad$ $TrB_{C_3}^{\ 0} = \{9, 14\}$

$TrS_{C_3}^{\ 1} = \{1, 2, 3, 4, 9, 11, 14, 16, 20\}$ $\qquad$ $TrB_{C_3}^{\ 1} = \{9, 14\}$

$rS_{C_3} = TrS_{C_3}^{\ 2} = \{1, 2, 3, 4, 9, 11, 14, 16, 20\}$

These three transform slices differ for the reading statements which are included in the recovered function, depending on what input variables are considered in the transform slicing criteria. However, all the transform slices have in common the exclusion of the loop statement which controls the processing of the entire student file.

In all the examples above, the exclusion of the conditional statements depends from the position of the output statements which have been selected as initial statements in the transform slicing criteria. However, in the case it would be not possible to find an output statement in the proper place, and the last program statement was instead selected as initial statement of a transform slicing criterion, then the extraneous conditional statements could not be eliminated.

3. Interprocedural Extraction Criteria

In this section the definitions given in Section 2 are extended to cover procedural programs where slices can cross the boundaries of procedure calls. We assume a language model in which parameters are passed by value-result and by reference, procedures can be nested and global variables are visible in the nested procedures. The model is sufficiently general to be usable with many programming languages by applying or restricting the assumptions. In the subsections below we give basic definitions and our rules for extracting functional components correctly.

3.1 Background

The following basic definitions are given to provide a common framework for the rules for interprocedural slicing. Interprocedural control flow graphs, interprocedural walks and interprocedural def/use graphs are defined as in [35] but we give a different interpretation for the variables defined and used by procedure calls. Weiser's extension to interprocedural slicing completes the basic definitions.

Definition 17: An *interprocedural control flow graph* $\mathcal{G}$ for a program is a tuple $(G_1, \ldots.. , G_k, CALL, RET)$ where $G_1, \ldots.. , G_k$ are control flow graphs representing program procedures, *CALL* is a set of call edges, and *RET* is a set of return edges satisfying the following conditions: (1) a *call edge* from a caller G_i to a callee G_j is of the form (n, n_I) where n is a procedure call of some N_{G_i}, and n_I is the initial node of some N_{G_j}; (2) a *return edge* from a callee G_j to a caller G_i is of the form (n_F, n) where n_F is the final node of some N_{G_j} and n is a procedure call of some N_{G_i}; (3) for each call edge (n, n_I) there is a return edge (n_F, n) such that n_I and n_F are the initial and final nodes of the same procedure; (4) there is a main procedure G_{main} whose two distinguished nodes are the distinguished nodes of $\mathcal{G}$: $n_{I\mathcal{G}}$ and $n_{F\mathcal{G}}$.

Definition 18: An *interprocedural walk W* in an interprocedural control flow graph $\mathcal{G} = (G_1, \ldots.., G_k, CALL, RET)$ is a sequence of nodes $n_1 n_2 \cdots n_l$, where $n_i \in (N_{G_1} \cup \cdots \cup N_{G_k})$ for $i = 1, \ldots, l$, $(n_j, n_{j+1}) \in (E_{G_1} \cup \cdots \cup E_{G_k} \cup CALL \cup RET)$, satisfying the following conditions:

(1) W contains the sequence $un_{IG}\mathcal{V}n_{FG}v$ where $G \in \mathcal{G}$ and $u \neq v$ iff $un_{IG} \subseteq \mathcal{V}$;

this condition guarantees that control flow from a procedure call will return only to it, i.e. calling context is saved.

(2) W does not contain a sequence $n_{FG}vn_{IG}$ where $G \in \mathcal{G}$ and $v \in (N_{G_1} \cup \cdots \cup N_{G_k})$;

this condition guarantees that control flow does not come back inside a procedure just after leaving it.

(3) W contains the sequence uwv where $G \in \mathcal{G}$, $(v, n_{IG}) \in CALL$, and $u \neq n_{FG}$ iff $w = n_{IG}$;

this condition guarantees that control flow goes inside a procedure unless it has just returned from it.

Definition 19: An *interprocedural def/use graph* is a quadruple $\Theta = (\mathcal{G}, \Sigma, D, U)$, where $\mathcal{G} = (G_1, \ldots , G_k, CALL, RET)$ is the interprocedural control flow graph representing a program, Σ is a finite set of symbols naming variables in the program, $D : (N_{G_1} \cup \cdots \cup N_{G_k}) \rightarrow \mathbb{P}(\Sigma)$, and $U : (N_{G_1} \cup \cdots \cup N_{G_k}) \rightarrow \mathbb{P}(\Sigma)$ are functions mapping the nodes of $\mathcal{G}$ in the set of variables which are defined or used in the statements corresponding to nodes.

With respect to definition 6, defined and used variables for procedure call statements must be added. Let n_{call} be a procedure call statement invoking a procedure G_j. A variable x is *defined* in n_{call} if the execution of G_j assigns a value to x, while a variable x is *used* in a statement n_{call} if the execution of G_j requires the value of x to be evaluated. An interprocedural data flow analysis is required to obtain the necessary summary information.

In [23], *potential data flows* among procedures are computed according to the visibility rules of the language model. The resulting sets of variables reflect the possibility that two procedures communicate through a variable which is located in their scope. To achieve a more precise definition from static analysis of source code another approach can be adopted, where *actual data flows* are derived, also in the presence of global variables and aliasing [11].

Here we assume we can obtain $U(n_{\text{call}})$, the set of variables used in a calling procedure and defined in the called procedure, and $D(n_{\text{call}})$, the set of variables defined in a calling procedure and used in the called procedure. In this way, global variables can be treated as additional parameters where formal and actual are the same thing. So, from now on, we will only discuss parameters.

Definition 20: Let $\Theta = (\mathcal{G}, \Sigma, D, U)$ be an interprocedural def/use graph, where $\mathcal{G} = (G_1, \ldots , G_k, CALL, RET)$. $SCOPE : \{G_1, \ldots, G_k\} \rightarrow \mathbb{P}(\Sigma)$ is a function mapping a program procedure in the set of variables which can be accessed from it.

Definition 21: Let $\Theta = (\mathcal{G}, \Sigma, D, U)$ be an interprocedural def/use graph, where $\mathcal{G} = (G_1, \ldots, G_k, CALL, RET)$, n a call to some procedure G_i, and $FNV : (N_{G_1} \cup \cdots \cup N_{G_k}) \rightarrow \mathbb{P}(\Sigma)$, a function mapping the nodes of $\mathcal{G}$ in the set of variables. $FNV(n)_{F \rightarrow A}$ means the substitution of formal for actual parameters in $FNV(n)$. $FNV(n_{IG_i})_{A \rightarrow F}$ means the substitution of actual for formal parameters in $FNV(n_{IG_i})$.

We assume we have a procedural program P represented by an interprocedural def/use graph $\Theta = (\mathcal{G}, \Sigma, D, U)$ where $\mathcal{G} = (G_1, \ldots , G_k, CALL, RET)$. Weiser's interprocedural slicing occurs in two steps. The former works as described in the previous section with only intraprocedural equations and summary data flow information for procedure calls. In the latter step, called and calling procedures are sliced with a new criterion. The two steps are repeated until there are no new procedures to be sliced.

The slicing criteria generated for encountered procedures differ according to whether the new procedure is a callee or a caller. In the former case, the new slicing criterion enables descent in the called procedure, while in the latter, a set of slicing criteria is generated to ascend to all callers.

Definition 22: Let G_i and G_j be two control flow graphs in $\mathcal{G}$ such that there exists a call edge (n, n_{IG_j}) and a return edge (n_{FG_j}, n) where $n \in N_{G_i}$ is a procedure call. If G_i is being sliced a *descending slicing criterion* for G_j is defined as

$$C = \langle\, n_{FG_j}\, , R_C(SUCC(n))_{F \rightarrow A} \cap SCOPE(G_j) \,\rangle$$

Definition 23: Let G_i and G_j be two control flow graphs in $\mathcal{G}$ such that there exists a call edge (n, n_{IG_j}) and a return edge (n_{FG_j}, n) where $n \in N_{G_i}$ is a procedure call. If G_j is being sliced an a*scending slicing criterion* for G_i is defined as

$$C = \langle\, n, R_C(n_{IG_j})_{A \rightarrow F} \cap SCOPE(G_i) \,\rangle$$

3.2 Interprocedural Extension for Transform Slicing

Let $\Theta = (\mathcal{G}, \Sigma, D, U)$ be an interprocedural def/use graph where $\mathcal{G} = (G_1, \ldots\ldots, G_k, CALL, RET)$, and P is a procedural program represented by Θ. Interprocedural slicing rules given in the previous subsection to generate ascending and descending slicing criteria are adopted for transform slicing with one amendment which we discuss below.

The problem to be solved is an imprecision of the Weiser's method due to the lack of a mechanism to account for the calling context of a called procedure. In [23], the calling-context problem is solved but the approach differs from ours because interprocedural slicing is dealt with as a reachability problem on a dependence graph. Since we use a data-flow equation approach, we use the definition of interprocedural walk given in [35], which is compatible with our representation of a program.

Amendment (for called procedures when transform slicing): Let G_{h}, G_{i} and G_{j} be three control flow graphs in $\mathcal{G}$ such that there exist two call edges (n, $n_{\mathrm{I}G_{\mathrm{j}}}$), (m, $n_{\mathrm{I}G_{\mathrm{j}}}$) and two return edges ($n_{\mathrm{F}G_{\mathrm{j}}}$, n), ($n_{\mathrm{F}G_{\mathrm{j}}}$, m) where $n \in \mathrm{N}_{G_{\mathrm{i}}}$, $m \in \mathrm{N}_{G_{\mathrm{h}}}$ are procedure calls. If G_{i} is being sliced the procedure call n causes the slice to descend into G_{j}; when the slice reaches $n_{\mathrm{I}G_{\mathrm{j}}}$ it ascends only following a valid interprocedural walk, i.e. it returns to G_{i} and not to G_{h}.

4. Using Transform Slicing for Building Reusable Assets

In order to be applied, transform slicing requires that a correct slicing criterion be formulated. The following problems have to be answered: how to get a list of expected functions to be recovered together with a partial specification in terms of input/output data, and how to cope with the difficulty in finding the last statement of an expected function, corresponding to the initial statement in the slicing criterion from which going backward in the source code.

4.1 Expected functions elicitation

Transform slicing requires the availability of knowledge about the application and programming domain. Domain knowledge suggests that some conceptually simple tasks are performed in the system and that these tasks are clearly defined at least in terms of their input and output data. Information can come from both static sources and dynamic sources. Static sources include the source code, the available documents related to the application, and standards. Dynamic sources include domain experts, developers, maintainers, end users and the direct interaction with the system itself.

In [2], the authors introduce the idea that, for data-oriented applications such as business applications, the reverse engineering process should include a data recovery phase before proceeding with the function recovery phase. The purpose of this data recovery phase is to produce a data model of the application system expressed using a hierarchical Entity-Relationship diagram and a data dictionary.

A method for data model construction was provided, based upon the use of a domain representation and the classification of source code variables. The domain representation contains domain entities, entity hierarchies, associative relationships, and entity attributes which define the application domain for a whole class of problems. The formalism used for this model is the same as the application data model, which is the end-product of the data recovery phase. They differ with respect to the level of abstraction used to describe the problem [9]. The domain representation is expressed at the conceptual level, which describes the problem in terms of a class of applications belonging to a certain domain, for example the banking domain. The application data model is expressed at the requirement level which provide greater details of a specific user problem belonging to a certain class of application, for example the XYZ Bank information system. The application data model is produced by extending the domain representation from the conceptual to the requirement level. As in [18], the domain representation acts as a scheme for driving the reverse engineering process and a template for organizing its results. Variable classification can make a distinction between variables which can be mapped to some object in the domain representation and variables which cannot, so that this mapping can be annotated in the data dictionary.

In [1], the authors propose five variable classification categories: basic conceptual data, derived conceptual data, control data, structure data and redundant data. Both basic and derived conceptual data can be mapped to an entity attribute in the application domain. They differ since conceptual derived data can be calculated from basic conceptual data or other conceptual derived data. This is an important distinction because the presence of derived data generates expectations on the existence of transform functions. These functions will have conceptual derived data as output, and basic or derived conceptual data as input. Recording this information in the data dictionary provides the specification of the expected functions to be extracted. Control data record a past event and are used to control the logic of a program. Also control data can help to specify the interface of expected functions, for example a cancellation flag could be considered the output of a function which logically deletes a record. Structure data are used to build more complex data structures. They can help to identify relationships between entities, as for example the presence of pointer to another data structure. Redundant data are aliases which must be reconnected to the original name.

Another useful classification [25] provides eight classification variables. Among these, the most important categories for deriving expected functions are domain variables, program variables, input variables and output variables. Domain variables, like conceptual data in the previous classification, can be mapped to objects in the application domain while program variables cannot because they implement concepts in the programming domain. Input variables are involved in input events such as reading from files or from the keyboard, while output variables are involved in output events such as writing to a file or to the screen. The classification of domain versus program variables combined with the classification of input versus output variables supplies a number of expected functions formulated in terms of input/output which could be extracted from the source code. For example, meaningful business functions producing external results from external inputs can be characterized by domain input variables and domain output variables. On the contrary, functions in the programming domain could be characterized by program input variables and output program variables.

During the classification activity, new entities, relationships, entity attributes and data dictionary entries are added to the initial domain representation which evolves to an application data model. At the end of the data recovery phase, the data dictionary will contain the description of the variables and the mapping between the model and the source code. A further step, combining information contained in the data dictionary with the functions found in the static sources or suggested by dynamic sources, provides a list of expected functions specifications with the following information:

- a function name
- a description of the function in free text format
- input parameter list (variables in the source code)
- output parameter list (variables in the source code)

4.2 Concept validation of transform slices

Transform slicing is a useful technique for extracting pieces of code which implement functional abstractions, but in addition to the input/output interface of the function, one needs to know the last statement of a function. This last statement must be specified in the transform slicing criterion as the statement from which slicing begin to go backward in the source code. As programs become larger and larger, this statement becomes more difficult to identify, requiring to read a lot of code.

To be realistically applicable with large programs, we need a technique which does not depend on statement numbers. Decomposition slice [22] satisfies this requirement because it depends from a variable but not from a statement number. A decomposition slice corresponds to the sets of all the instructions which contribute to the value of a variable *v* at all the points in a program where the variable becomes visible outside the program. The decomposition slice is defined as the union of all the program slices with the output statements of v and the last program statement specified in the slicing criterion. The last statement of a program is included to specify variables which do not compare in output statements and to capture any computation of a variable performed after its last output.

However, this approach cannot be totally accepted for recovering reusable functions. Extracting the implementation of a functional abstraction by making the union of a collection of transform slices have three weak points. First, we might obtain a functional component which computes more times the same result because ill-structured programs often contain duplicated code and even different implementations of a same function. Second, we loose the confidence of obtaining cohesive functions which implement a single task because if an output variable name is used for more different purposes this can lead to extract all the functions sharing this same variable name. Third, as an effect of the inclusion rule for conditional statements, transform slicing from the last statement program usually includes more conditional statements than transform slicing from output statements. As a result, the union of transform slices will contain more conditional statements than necessary to the implementation of the functional abstraction because it throws away information related to the program position of the slicing criterion statement.

Although it is not possible to obtain reusable components simply as the union of transform slices, we can incorporate the approach behind decomposition slicing by providing a process which require a user validation of the extracted functions. The process for extracting functional components is shown in Figure 5. The process receives in input a program and a list of expected functions specifications, including a meaningful name, a function description and their input/output data. The process produces in output a list of functional components which implement the expected functions and have been elected to be reused. The description of the process uses the following functions:

- name (fn) returns the name for function fn
- description (fn) returns the textual description for function fn
- input-to (fn) returns the input parameters for function fn
- output-from (fn) returns the output parameters for function fn
- output (v) returns the set of statements that output variable v
- transform-slice (statement, input variables, output variables) returns a transform slice using the transform slicing criterion < statement, input variables, output variables >
- remove-duplicates (slices) returns a set of distinct slices
- validate-concept (slice, concept) returns true if the slice implements the given concept

This last function requires the user interaction to elect the transform slice which implements the functional abstraction among the candidates obtained with a different statement in the slicing criterion. While concept assignment [8] consists in trying to associate a human-oriented concept to unknown code segments, this is a concept validation task because code segments are filtered through a given human-oriented concept. Although the process requires a frequent validation to choose the right slice among the candidates, the user is asked to read small similar pieces of code compared to the amount of code necessary to identify the last statement of the expected function.

```
for each fn in expected-functions
    reuse-candidates = ∅
    name = name (fn)
    concept = description (fn)
    inpvars = input-to (fn)
    outvars = output-from (fn)
    for each stmt in output (outvars) ∪ {last-stmt}
        reuse-candidates = reuse-candidates ∪ transform-slice (stmt, inpvars, outvars)
    reuse-candidates = remove-duplicates (reuse-candidates)
    for each slice in reuse-candidates
        if validate-concept (slice, concept) then
            reuse-elected = reuse-elected ∪ (name, concept, inpvars, outvars, slice)
return reuse-elected
```

Fig. 5. Extraction of reusable functions

5. Related Work

The production of reusable components from legacy systems is not unique to slicing. A pioneering work in this field was Care [10], a tool based on a metric model of reusability. Looking at the source code, Care identifies routines or units which satisfy the metric values typical of components with a high frequency of reuse. However, components with these characteristics should have a high cohesion and a low coupling, which is seldom the case with legacy systems.

A great deal of research has focused on recognizing data abstractions by means of static code analysis. Components which implement data abstractions are typically recovered by aggregating existing routines around a group of data without modifying the statements inside of those routines [13, 14, 19, 32, 34]. Although this list is not exhaustive, the approaches differ as regards the aggregation criteria. However, in this case too, if the original application was not structured according to functional decomposition, the functional abstractions are not recognizable and the success of the modularization approach can be compromised.

Aggregation methods based on call graph analysis have been applied to cluster existing routines to form functions at a higher level of abstraction [15, 16]. The extracted clusters must be carefully read by domain-expert software engineers to identify what function they implement. However, when applied to large systems with many candidate clusters, the task of concept assignment [8] is difficult and time-consuming. On the contrary, the concept validation of transform slices, used in our method, requires an extracted component is read only to confirm that it is the implementation of a given expected function.

Conditioned slicing have been proposed to decompose modules which perform multiple unrelated functions or to isolate subfunctions executed under different condition branches [12, 26]. However, conditioned slicing requires the additional knowledge of a precondition or a triggering condition of the functional abstraction is searched for. If such a knowledge is not readily available, it might require to read a lot of code. On the contrary, transform slicing is able to decompose unrelated functions which are activated by a function code or to discard precondition tests without the need to specify the binding condition but simply analyzing the dependencies between any conditional statement and the statement in the slicing criterion.

The identification of functional abstractions in the source code can be seen as a particular problem of program segmentation. Segmenting a program means breaking it into chunks of code which are easier to manage. When the chunks relate to a unique functional behavior, they are the manifestation of programming plans [42]. While plans are abstract structures which programmers use as a programming template, the manifestation of plans in the programs is often delocalized [31] because they are realized by statements which are non-sequential. The usefulness of a segmentation technique increases the more the plans are scattered in the text of the program. In [39], plans are recovered in the form of clichés, by analyzing a flow graph which stores program information. Program segments, another name for the manifestation of plans, are recovered in [28] by means of concept recognition. Both approaches are based on the assumption that plans are implemented as stereotypical coding patterns. The search is driven by a *prescriptive* specification of the plan to be recovered, stating *how* the plan could be implemented. On the contrary, our slicing approach for the recovery of reusable components is descriptive, because the search is guided by a partial specification of what the function is supposed to do, in terms of its input/output interface. This specification is used for identifying those parts of the program which use input data to produce output data.

While the methods discussed above use static code analysis, dynamic methods have also been applied to identify reusable functions [3, 40, 44]. Dynamic methods are based on instrumenting a system and executing it with a baseline of test cases. Although there is no guarantee that each test case drives a single functionality, a large set of test cases improves the precision. Our slicing technique could be used in a complementary way, to achieve correct identification of functions after dynamic slicing.

6. Conclusions and Future Directions

We defined a slicing technique, called transform slicing, which is different from conventional Weiser's slicing. While conventional slicing was proposed for program understanding and debugging purposes, these new technique is designed to extract reusable functions from existing ill-structured programs. This different goal requires a redefinition of the notion of program slice so as to obtain code segments which actually implement the specified functional abstractions.

Transform slicing is language-independent but its application to realistic programs requires further research to solve language-specific problems, for example aliasing [37].

A weakness of the method presented is the efficiency of the algorithm. Since the transform slicing algorithm is based on def/use graphs, the worst case running times are $O(n\ e\ log(e))$ for producing a single transform slice, and $O(n^2\ e\ log(e))$ for obtaining the candidates of an expected function. Other slicing algorithms, based on program dependence graphs [20] or system dependence graphs [23] can obtain slices in linear time. However, these kinds of program representation require that slicing starts where the slice variable is defined or used, while our method for obtaining candidate functions requires that slicing starts from the last program statement too. Further research is needed improve the efficiency of our algorithm without burdening the human intervention.

Currently, we are developing an interactive prototype system according to the extraction method presented in this paper. The tool is going to analyze COBOL programs but we intend to extend its scope to other imperative languages like C and Pascal.

The underlying method and the tool based on it need to be empirically evaluated. We have to assess the completeness and the accuracy of the extracted slices. Completeness is the property of a slice of including all the statements needed to implement the related functional abstraction. Accuracy is the property of a slice of not including statements which are extraneous to the related functional abstraction. For the purpose of understanding, the extracted functions should be characterized, including metrics such as the size of slices, the percentage of the original module size, the length of contiguous statements, and the number of contiguous code fragments. Finally, a controlled experiment might be designed to compare the approach based on finding the last statement of an expected function directly in the source code, and the approach based on the the concept validation of the automatically extracted candidate functions.

Acknowledgements

We would like to thank Victor R. Basili, Gianluigi Caldiera, and Koji Torii for their helpful comments and suggestions on a previous version of this paper.

References

[1] F. Abbattista, G. M. G. Fatone, F. Lanubile, and G. Visaggio, “Analyzing the application of a reverse engineering process to a real situation”, Proceedings of the 3rd Workshop on Program Comprehension, Washington D.C., November 1994, pp.62-71.

[2] F. Abbattista, F. Lanubile, and G. Visaggio, “Recovering conceptual data models is human-intensive”, *Proceedings of the Fifth International Conference on Software Engineering and Knowledge Engineering*, San Francisco, California, 1993, pp.534-543.

[3] H. Agrawal, and J. R. Horgan, “Dynamic program slicing”, *Proceedings of the ACM SIGPLAN’90 Conference on Programming Language Design and Implementation*, June 1990, pp.246-256.

[4] H. Agrawal, J. R. Horgan, E. W. Krauser, and S. A. London, “Incremental regression testing”, *Proceedings of the Conference on Software Maintenance*, Montreal, Quebec, Canada, September 1993, pp.348-357.

[5] V. R. Basili, G. Caldiera, F. McGarry, R. Pajerski, G. Page, and S. Waligora, “The Software Engineering Laboratory: an operational software experience factory”, *Proceedings of the 14th International Conference on Software Engineering*, Australia, 1992, pp.370-381.

[6] J. Beck, and D. Eichmann, “Program and interface slicing for reverse engineering”, *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, MD, 1993, pp.54-63.

[7] K. Bennett, “Legacy systems: coping with success”, *IEEE Software*, vol.12, no.1, January 1995, pp.19-23.

[8] T. J. Biggerstaff, B. G. Mitbander, and D. E. Webster, “Program understanding and the concept assignment problem”, *Communications of the ACM*, vol.37, no.5, 1994, pp.72-83.

[9] E. J. Byrne, “A conceptual foundation for software reengineering”, *Proceedings of the Conference on Software Maintenance*, Orlando, Florida, 1992, pp.226-235.

[10] G. Caldiera, and V. R. Basili, “Identifying and qualifying reusable software components”, *IEEE Computer*, February 1991, pp.61-70.

[11] G. Canfora, and A. Cimitile, Reverse engineering and intermodular data flow: a theoretical approach”, *Software Maintenance: Research and Practice*, vol.4, 1992, pp.37-59.

[12] G. Canfora, A. Cimitile, A. De Lucia, and G. A. Di Lucca, “Software salvaging based on conditions”, *Proceedings of International Conference on Software Maintenance*, Victoria, Canada, September 1994, pp.424-433.

[13] G. Canfora, A. Cimitile, and M. Munro, “A reverse engineering method for identifying reusable abstract data types”, *Proceedings of the Working Conference on Reverse Engineering*, Baltimore, MD, 1993, pp.73-82.

[14] G. Canfora, A. Cimitile, M. Tortorella, and M. Munro, “A precise method for identifying reusable abstract data type in code”, *Proceedings of International Conference on Software Maintenance*, Victoria, Canada, 1994, pp.404-413.

[15] S. C. Choi, and W. Scacchi, “Extracting and restructuring the design of large systems”, *IEEE Software*, January 1990, pp.66-71.

[16] A. Cimitile, and G. Visaggio, “Software salvaging and call dominance tree”, unpublished DIS internal report, 1992. To appear in *The Journal of Systems and Software*.

[17] F. Cutillo, F. Lanubile, and G. Visaggio, “Extracting application domain functions from old code: a real experience”, *Proceedings of the 2nd Workshop on Program Comprehension*, Capri, Italy, July 1993, pp.186-191.

[18] J. M. DeBaud, B. Moopen, and S. Rugaber, “Domain analysis and reverse engineering”, *Proceedings of International Conference on Software Maintenance*, Victoria, Canada, 1994, pp.326-335.

[19] M. F. Dunn, and J. C. Knight, “Automating the detection of reusable parts in existing software”, *Proceedings of the 15th International Conference on Software Engineering*, Baltimore, MD, 1993, pp.381-390.

[20] J. Ferrante, K. Ottenstein, and J. Warren, “The program dependence graph and its use in optimization”, *ACM Transactions on Programming Languages and Systems,* vol.9, no.3, July 1987, pp.319-349.

[21] P. Fiore, F. Lanubile, and G. Visaggio, “Analyzing empirical data from a reverse engineering project”, *Proceedings of the 2nd Working Conference on Reverse Engineering*, Toronto, Ontario, Canada, July 1995.

[22] K. B. Gallagher and J. R. Lyle, “Using program slicing in software maintenance”, *IEEE Transactions on Software Engineering*, vol.17, no.8, August 1991, pp.751-761.

[23] S. Horwitz, T. Reps, and D. Binkley, “Interprocedural slicing using dependence graphs”, *ACM Transactions on Programming Languages and Systems*, vol.12, no.1, January 1990, pp.26-60.

[24] J. Jiang, X. Zhou, and D. J. Robson, “Program slicing for C - the problems in implementation”, *Proceedings of Conference on Software Maintenance*, Sorrento, Italy, 1991, pp.182-190.

[25] J. K. Joiner, W. T. Tsai, and X. P. Chen, “Data-centered program understanding”, *Proceedings of International Conference on Software Maintenance*, Victoria, Canada, 1994, pp.272-281.

[26] H. S. Kim, Y. R. Kwon, and I. S. Chung, “Restructuring programs through program slicing”, *International Journal of Software Engineering and Knowledge Engineering*, vol.4, no.3, 1994, pp.349-368.

[27] B. Korel, and J. Laski, “Dynamic program slicing”, *Information Processing Letters,* vol.29, no.3, October 1988, pp.155-163.

[28] W. Kozaczynski, J. Ning, and A. Engberts, “Program concept recognition and transformation”, *IEEE Transactions on Software Engineering*, vol.18, no.12, December 1992, pp.1065-1075.

[29] C. W. Krueger, “Software reuse”, ACM Computing Surveys, vol.24, no.2, June 1992, pp.131-183.

[30] F. Lanubile, and G. Visaggio, “Function recovery based on program slicing”, *Proceedings of Conference on Software Maintenance,* Montreal, Quebec, Canada, 1993, pp.396-404.

[31] S. Letovski, and E. Soloway, “Delocalized plans and program comprehension”, *IEEE Software,* May 1986, pp.198-204.

[32] S. Liu, and N. Wilde, “Identifying objects in a conventional procedural language: an example of data design recovery”, *Proceedings of Conference on Software Maintenance*, San Diego, CA, 1990, pp.266-271.

[33] P. E. Livadas, and S. D. Alden, “A toolset for program understanding”, *Proceedings of the 2nd Workshop on Program Comprehension,* Capri, Italy, 1993, pp.110-118.

[34] P. E. Livadas, and P. K. Roy, “Program dependency analysis”, *Proceedings of Conference on Software Maintenance*, Orlando, FL, 1992, pp.356-365.

[35] J. P. Loyall, “Using dependence analysis to support the software maintenance process”, *Proceedings of Conference on Software Maintenance*, Montreal, Quebec, Canada, 1993, pp.282-291.

[36] J. R. Lyle, and M. D. Weiser, “Automatic program bug location by program slicing”, in *Proceedings of the 2nd International Conference on Computers and Applications*, Peking, China, June 1987, pp.877-882.

[37] H. D. Pande, W. A. Landi, and B. G. Ryder, “Interprocedural def-use associations for C systems with single level pointers”, *IEEE Transactions on Software Engineering*, vol.20, no.5, May 1994, pp.385-403.

[38] A. Podgurski, and L. A. Clarke, “A formal model of program dependences and its implications for software testing, debugging, and maintenance”, *IEEE Transactions on Software Engineering*, vol.16, no.9, September 1990, pp.965-979.

[39] C. Rich, and L. Wills, “Recognizing a program's design: A graph parsing approach”, *IEEE Software*, January 1990, pp.82-89.

[40] H. Ritsch, and H. M. Sneed, “Reverse engineering program via dynamic analysis”, *Proceedings of the Working Conference on Reverse Engineering*, Baltimore, MD, 1993, pp.192-201.

[41] S. Rugaber, K. Stirewalt, and L. M. Wills, “The interleaving problem in program understanding”, *Proceedings of the Second Working Conference on Reverse Engineering*, Toronto, Canada, 1995, pp.166-175.

[42] E. Soloway, and K. Ehrlich, “Empirical studies of programming knowledge”, *IEEE Transactions on Software Engineering*, vol.SE-10, no.5, 1984, pp.595-609.

[43] M. Weiser, “Program slicing”, *IEEE Transactions on Software Engineering*, vol.SE-10, no.4, July 1984, pp.352-357.

[44] N. Wilde, J. A. Gomez, T. Gust, and D. Strasburg, "Locating user functionality in old code", *Proceedings of Conference on Software Maintenance*, Orlando, FL, IEEE Computer Society Press, 1992, pp.200-205.