PROGRAM SLICING IN KNOWLEDGE-BASED LOOP ANALYSIS

Marwa A. Sorour and Salwa K. Abd-El-Hafiz

Engineering Mathematics Department, Faculty of Engineering,
Cairo University, Giza, Egypt.

ABSTRACT

Loops used in a program have an important effect on the ability to understand it. This is attributed to the inherent difficulty in reasoning about loops. In this paper, we present a knowledge-based approach for the automation of loop understanding and documentation. This approach uses a well-known previously designed decomposition technique, called program slicing, to mechanically decompose loops by analyzing their control and data flow. Each slice of the loop results in isolating the effect of using this loop on computing a single variable. The resulting slices are then analyzed by utilizing patterns, called plans, stored in a knowledge base, to generate their first-order predicate logic annotations. Because of the mathematical basis of these annotations, correctness conditions can be stated and verified if desired, using the axiomatic correctness approach. Finally, we present the results of a case study performed to evaluate our proposed analysis approach. In this study, we design a set of plans by analyzing loops in a real and existing program. This initial set of plans is then used to analyze loops in other programs of reasonable sizes and practical value. Results concerning the utilization of plans are given and discussed. These results generally show a good usability of the knowledge base beyond the original program.

Keywords: Loop analysis, Program slicing, Loop understanding, Knowledge base, Reverse engineering.

INTRODUCTION

Program understanding plays an important role in nearly all software-engineering tasks. It is vital to maintenance, documentation, debugging, reuse and testing. Without an adequate and deep understanding of a program, it is impossible to maintain it effectively. Program understanding is indispensable for the reuse of code components, because they can not be reused without a clear understanding of what they do. Testing, debugging and documentation also require programmers to read and understand programs carefully.

Because of the importance of program understanding, considerable research has been concerned with its automation. Some research efforts were directed towards the automation of program analysis and understanding in general. Due to the reported evidence that loops used in a program have an important effect on its understandability [1], other research focused on loop analysis and understanding. Consequently, many encouraging and useful approaches are available. These approaches can be classified into two broad categories : algorithmic and knowledge-based approaches.

The algorithmic approaches [2-4] generate formal and semantically sound documentation (e.g., by using first-order predicate logic) that annotates programs according to the formal semantics of a specific model of correctness (e.g., the axiomatic correctness approach [5, 6] and Mills functional correctness approach [7]). Yet, a common limitation to these approaches, is that they rely on the user to provide the loop annotations. They offer mechanical assistance only in proving the correctness of these annotations and in producing the specifications of the whole program.

Knowledge-based approaches [8-12] are a good solution to the limitation of the algorithmic approaches. They modularize experts' knowledge in the form of plans that can be accessed mechanically. In these approaches, the generation of a component's documentation usually involves two main tasks: the recognition of stereotyped parts in the program and deriving their annotations using plans stored in the knowledge base. Yet, some of these approaches produce documentation which is more or less in the form of natural text [9,10,12]. Such informal documentation gives expressive and intuitive descriptions of the code. However, there is no semantic basis that makes it possible to determine whether the documentation has the desired meaning. This lack of firm semantic basis makes informal natural documentation inherently ambiguous.

We focus, in this paper, on the analysis by decomposition approaches which represent good examples of the knowledge-based approaches [8,13,14]. Analysis by decomposition approaches breaks up the program into smaller more tractable parts. These smaller parts are then matched against the knowledge base to derive their annotations (documentation). The understanding of these smaller parts yields an understanding of the whole program.

More specifically, we present a knowledge-based approach for the automation of loop understanding. It is motivated by the idea of analysis by decomposition [15,16]. We investigate the effect of using a well-known previously designed decomposition technique called program slicing on knowledge-based loop understanding. A program slice consists of the parts of the program that potentially affect the values computed at some point of interest. Program slices are computed by analyzing the program's data flow and control flow. The resulting loop slices are then analyzed by using plans stored in a knowledge base to deduce their annotations. Finally, we will test and evaluate our proposed approach on real and existing programs.

The next section of this paper provides the necessary background for program slicing. The section to follow describes the structure and design of our knowledge base, and how the plans are hierarchically arranged. Then, we describe the design and structure of our proposed prototype tool, followed by presenting an evaluation of our proposed approach. Results concerning the utilization of the designed plans in analyzing loops in different fields are discussed. Finally, conclusions and future research directions are given in the last section.

PROGRAM SLICING

Program slicing is a method for decomposing a program into pieces of code based on data flow and control flow information. A program slice consists of the parts of a program that (potentially) affect the values computed at some point of interest. Such a point of interest is referred to as a slicing criterion, and is typically specified by a location in the program in combination with a subset of program's variables. The task of computing program slices is called program slicing. The original definition of a program slice was presented by Weiser [16], and then various slightly different notions of program slices were presented as well as a number of methods to compute them [17-19].

Program slicing can help a programmer understand complicated code and can aid in debugging [20]. It has been used in testing, modification [21] and maintenance [22]. Slicing has also been used as a structural method for isolating functionalities in large programs [3]. An important distinction is that between a static and a dynamic slice. Static slices are computed without making assumptions regarding a program's input, whereas the computation of dynamic slices relies on a specific test case [18]. Throughout this section, we focus on static slicing as it is suitable for program analysis for the sake of understanding. We introduce some basic slicing definitions. We define how to perform intraprocedural slicing (i.e., slicing a program that consists of a single monolithic procedure). Due to space limitations, we do not describe interprocedural slicing and slicing in the presence of composite data structures and pointers. For more details on the different aspects of program slicing, refer to the numerous literature that exists on this topic (see for example References 16, 17, 19, 20, 23, 24, 25.

Basic Definitions

Definition 1: A *digraph* G is a tuple (N, E), where N is a set of nodes and $E \subseteq N \times N$ is a set of edges. Given an edge (m, n), m is said to be *immediate predecessor* of n, or $m \in \mathrm{IMP}(n)$, while n is said to be *immediate successor* of m, or $n \in \mathrm{IMS}(m)$. A *path* p from m to n of length k is an ordered set of nodes $(n_0, .., n_k)$ such that $n_0 = m$, $n_k = n$, and $\forall i$, $0 \leq i \leq k-1 : (n_i, n_{i+1}) \in \mathrm{E}$. □

Definition 2: A *flow graph FG* is a triple (N, E, n_0), where (N, E) is a digraph and $n_0 \in N$ is such that $\forall n \in N$ there is a path $(n_0, \ldots, n)$. n_0 is called the *initial node*. Given $m, n \in N$, m *dominates* n, or $m \in \mathrm{DOM}(n)$, if m is on every path $(n_0, \ldots, n)$. Given $m, n \in N$, m is the *nearest dominator* of n, or $m = \mathrm{NDOM}(n)$, if $m \in \mathrm{DOM}(n)$ and $\forall d \in \mathrm{DOM}(n)$: $d \in \mathrm{DOM}(m)$. □

Definition 3: A *hammock graph HG* is a quadruple (N, E, n_0, n_e), where (N, E, n_0) and (N, E^{-1}, n_e) are both flow graphs. n_e is called the *final node*. Given $m, n \in N$, m *inverse dominates* n, or $m \in \mathrm{IDOM}(n)$, if m is on every path $(n, \ldots, n_e)$. Given $m, n \in N$, m is the *nearest inverse dominator* of n, or $m = \mathrm{NIDOM}(n)$, if $m \in \mathrm{IDOM}(n)$ and $\forall d \in \mathrm{IDOM}(n)$: $d \in \mathrm{IDOM}(m)$. □

Definition 4: A one entry/one exit program P can be modeled as hammock graph, whose nodes are the program statements, the edges are given by the control flow, the entry point is the initial node, and the exit point is the final node. This gives a definition of a program's control-flow graph. □

Definition 5: Given $n \in N$, the *statements influenced by* n, or $\mathrm{INFL}(n)$, are the set of nodes which are on a path $(n, \mathrm{NIDOM}(n))$, excluding the endpoints n and $\mathrm{NIDOM}(n)$. When $\mathrm{IMS}(n)$ contains only one element or is empty then $\mathrm{INFL}(n) = \phi$. □

Definition 6: A node i in the CFG is *post-dominated* by a node j if all paths from i to n_e pass through j, where n_e is the final node. □

Definition 7: Let V be the set of variables in a program P, $\mathrm{REF}(i) \subseteq V$ are the variables used at instruction (node) i, and $\mathrm{DEF}(i) \subseteq V$ are the variables modified at instruction (node) i, i.e., whose values are changed as an effect of the instruction execution. □

Intraprocedural Slicing

The slice of a program, P, with respect to a program location, n, and a subset of the program's variables, V, consists of all statements and predicates of the program that might affect the values of variables in V at location n [16,26]. In Weiser's terminology [16], a slicing criterion of a program P is defined as a tuple $<n, V>$, where n is a statement in P and V is a subset of the variables in P. The slicing algorithm determines consecutive sets of relevant variables from which sets of relevant statements are derived. For each statement in P there will be some set of variables whose values can affect a variable observable at the slicing criterion. The set of such relevant variables at statement i is denoted by $\mathrm{R}^0_{\mathrm{C}}(i)$, and defined later.

The statements included in the slice by $\mathrm{R}^0_{\mathrm{C}}(i)$, $i = 1, \ldots, n$ are denoted by $\mathrm{S}^0_{\mathrm{C}}$. The superscript 0 indicates how indirect the relevance is, higher valued superscripts are defined later. Throughout this subsection, we will use the program in Figure 1 as an example program.

Definition 8: Let $C = <n, V>$ be a slicing criterion. Then $R^0_C(i)$ = all variables υ such that either:

1. $i = n$ and υ is in V, or
2 i is an immediate predecessor of a node m such that either:
   a) υ is in $\mathrm{REF}(i)$ and there is some variable w in both $\mathrm{DEF}(i)$ and $\mathrm{R}^0_{\mathrm{C}}(m)$, or
   b) υ is not in $\mathrm{DEF}(i)$ and υ is in $\mathrm{R}^0_{\mathrm{C}}(m)$. □

$R_C^0(i)$ represents the set of directly relevant variables when program execution is at instruction i (node i in the CFG). The recursion is over the length of paths to reach node i, where (1) is the base case. The computation of $R_C^0(i)$, $i = 1, \ldots, n$ starts with the initial values $R_C^0(n) = V$ and $R_C^0(i) = \phi$ for any node $i \neq n$. The computation of case (2a) says that if w is a relevant variable at the node following i and w is given a new value at node i, then w is no longer relevant and all variables used to define w's value are relevant. Case (2b) says that if a relevant variable at an immediate successor of node i is not given a value at node i, then it is still relevant at node i.

The statements included in the slice by $R_C^0(i)$, $i = 1, \ldots, n$ are denoted by S_C^0. S_C^0 is defined by $S_C^0 = \{ i \mid \mathrm{DEF}(i) \cap R_C^0(\mathrm{IMS}(i)) \neq \phi\}$. S_C^0 includes those statements whose execution can directly influence the values of relevant variables for each instruction of the program.

For the program shown in Figure 1 and slicing criterion <10, {product}>, Table 1 summarizes the DEF, REF sets and the sets of directly relevant variables at each node. After the computation of R_C^0, we obtain $S_C^0 = \{2,4,7,8\}$. Note that S_C^0 does not include any indirect effects on the slicing criterion. Generally any branch statement which can choose to execute some statement in S_C^0 should also be included in the slice. That is, statement 5 of our example should be included in the slice. The following definitions explain how to perform this modification.

<u>Definition 9</u>: $B_C^0 = \{b \mid \mathrm{INFL}(b) \cap S_C^0 \neq \phi\}$. □

B_C^0 includes all those conditional statements which can choose to execute or not execute some statements in S_C^0. Given a conditional statement a branch statement criterion can be issued: BC(b) = <b, REF(b)>. To include all indirect influences, the statements with direct influence on B_C^0 must now be considered, and then the branch statements influencing those new statements, etc., The full definition of the influence at level $j + 1$ is the following.

<u>Definition 10</u>: $\forall\, j \geq 0$:

$$R_C^{j+1}(i) = R_C^j(i) \bigcup_{b \in B_C^j} R_{BC(b)}^0(i).$$

$$S_C^{j+1} = \{i \mid \mathrm{DEF}(i) \cap R_C^{j+1}(\mathrm{IMS}(i)) \neq \phi\} \cup B_C^j.$$

$$B_C^{j+1} = \{b \mid \mathrm{INFL}(b) \cap S_C^{j+1} \neq \phi\}. \;\square$$

The base case is S_C^0, R_C^0 and B_C^0 the set of conditional statements that are indirectly relevant due to the influence they have on nodes i in S_C^0. $R_C^{j+1}(i)$ contains variables in $R_C^j(i)$. In addition, it contains variables that are directly relevant with respect to all slicing criteria in BC(b), $\forall b \in B_C^j$. For S_C^{j+1}, a set B_C^{j+1} is calculated, to include the statements which control the execution S_C^{j+1}. The R_C^{j+1} and S_C^{j+1} are non-decreasing subsets, and are bounded above by the set of program's variables and statements respectively. In other words, the iteration will stop when: $\forall i$: $R_C^{j+1}(i) = R_C^j(i)$ and $S_C^{j+1} = S_C^j = S_C$.

For the example shown in Figure 1, and slicing criterion <10, {product}>, S_C^0 and R_C^0 were computed and their values were shown in Table 1. We obtain $B_C^0 = \{5\}$, and BC(5) = < 5,{ i, n }>. We calculate $R_{BC(5)}^0$. Then R_C^1 is calculated as the union of R_C^0 and $R_{BC(5)}^0$. We obtain $S_C^1 = \{1, 2, 4, 5, 7, 8\}$ and B_C^1 evaluates to ϕ. So the process of calculating S_C terminates.

```
Program Main
    begin
(1)    read(n);
(2)    i := 1;
(3)    sum := 0;
(4)    product := 1;
(5)    while i < = n do begin
(6)      sum := sum + i;
(7)      product := product * i;
(8)      i := i + 1
       end;
(9)    write(sum);
(10)   write(product)
     end.
```

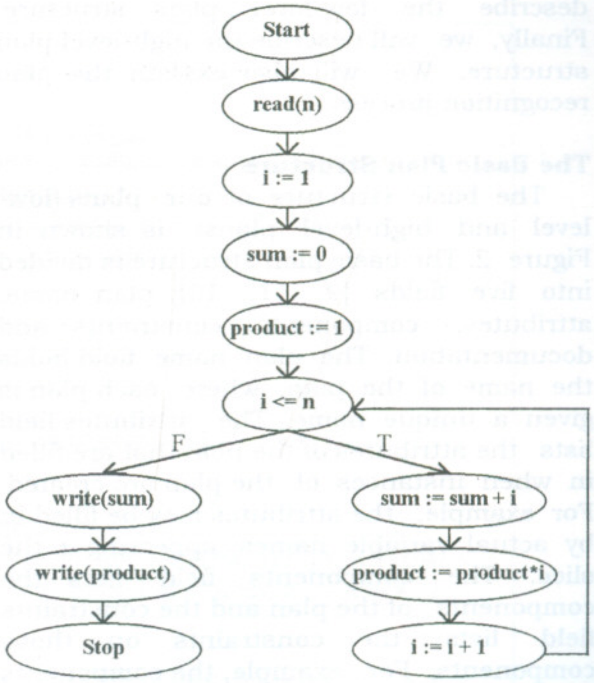

Figure 1 An example program and its control flow graph.

Table 1 Slicing results for the example program of Figure 1 and slicing criterion <10, {product}>.

No de	DEF	REF	R_C^0	INFL	$R_{BC(5)}^0$	R_C^1
1	{n}	ϕ	ϕ	ϕ	ϕ	ϕ
2	{i}	ϕ	ϕ	ϕ	{n}	{n}
3	{sum}	ϕ	{i}	ϕ	{i, n}	{i, n}
4	{product}	ϕ	{i}	ϕ	{i, n}	{i, n}
5	ϕ	{i, n}	{product, i}	{6, 7, 8}	{i, n}	{product, i, n}
6	{sum}	{sum, i}	{product, i}	ϕ	{i, n}	{product, i, n}
7	{product}	{product, i}	{product, i}	ϕ	{i, n}	{product, i, n}
8	{i}	{i}	{product, i}	ϕ	{i, n}	{product, i, n}
9	ϕ	{sum}	{product}	ϕ	ϕ	{product}
10	ϕ	{product}	{product}	ϕ	ϕ	{product}

KNOWLEDGE BASE DESIGN

Our approach to construct the knowledge base, is to organize our plans in a hierarchy [9, 11, 27]. At the lowest level — the leaf nodes — are plans representing source code constructs, and at higher levels — non-leaf nodes — are plans inferred from low-level plans. Corresponding to this hierarchy, we have two plan categories: low-level plans and high-level plans, respectively.

The understanding process is approached from the perspective of plan recognition. This process is executed in two steps. First, one identifies all instances of low-level plans in a given slice. Second, the identified low-level plans are composed to conform to the hierarchical structure of the knowledge base and to move towards a root node that represents a goal. In this section, we describe the design details of our hierarchical knowledge base. First, we describe the basic plan structure, then we describe the low-level plan structure. Finally, we will describe the high-level plan structure. We will also explain the plan recognition process.

The Basic Plan Structure

The basic structure of our plans (low-level and high-level plans) is shown in Figure 2. The basic plan structure is divided into five fields [9, 11, 13]; plan name, attributes, components, constraints and documentation. The plan name field holds the name of the plan, where each plan is given a unique name. The attributes field lists the attributes of the plan that are filled in when instances of the plan are created. For example, the attributes may be filled in by actual variable names appearing in the slice. The components field lists the components of the plan and the constraints field lists the constraints on those components. For example, the components may be templates of source code statements and the constraints may be on the ordering of those statements and on sharing of information among them. An instance of the plan is recognized when all of its components have been recognized without violating the constraints. The documentation field contains the information necessary to generate documentation in a formal specification language. First-order predicate calculus is used to generate the documentation in a Hoare style [6]. This is maintained in the precondition and invariant parts of the documentation. The precondition and invariant parts are patterns to be instantiated by the plan's attributes. In this section, we give two plan examples. We simplified the representation of these plans for demonstration purposes.

Low-Level Plans

Low-level plans have the generic template structure shown in Figure 2. The components are template components to be matched against the actual code of the slice. They can match slices written in Pascal. The template components do not contain the Pascal delimiter ';', as it does not affect the matching results. Low-level plans can be further specialized based on values that instantiate the plan's attributes. Figure 3 shows an example low-level plan, LPi. LPi represents a template plan for bubbling down the maximum or minimum element in an array. LPi has the attributes array#, index#, rop#, final#, cop# which represent an array variable, an index variable, a relational operator, the limit of the array range, and another relational operator used for comparison, respectively.

The components of the plan in Figure 3 are template components that are matched against the slice. The subscript '$_0$' is used to denote initial values and the suffix '#' is used to indicate the terms and attributes that must be instantiated with actual values in the code. We have constraints on the allowable values of the components. For example, rop# must be instantiated to a value in the set {<=, <} and cop# must be instantiated to a value in the set {<, <=, >, >=}. The last constraint is on the ordering of the components. The components should be ordered in the same way they are listed in the components field. This is not necessarily an immediate precedence relation.

Plan name	unique plan name
Attributes	attributes to be instantiated when an instance of the plan is created
Components	components of the plan
Constraints	constraints on the components
Documentation	
Precondition	a pattern when instantiated gives a precondition
Invariant	a pattern when instantiated gives an invariant

Figure 2 A generic plan template.

Plan name Lpi

Attributes array#, index#, rop#, final#, cop#.

Components

```
1. index# := index0#
2. while index# rop# final# do begin
3.     if array#[index#] cop# array#[index# + 1] then begin
4.         temp# := array#[index#]
5.         array#[index#] := array#[index# + 1]
6.         array#[index# + 1] := temp#
       end
7.     index# := index# + 1
   end
```

Constraints

1. rop# $\in\{<=, <\}$.
2. cop# $\in\{<, <=, >, >=\}$.
3. Component i precedes component i +1 $\forall$ i : 1 to 6.

Documentation

LPi.i cop# $\in$ {>=, >}.

Precondition $\text{index}_0\#$ rop# SUCC(final#).

Invariant $\text{index}_0\# \le \text{index}\#$ rop# SUCC(final#) $\wedge$ PERM(array#, $\text{array}_0\#$) $\wedge$ array#[index#] $\ge$ MAX(array# [ind], ind = $\text{index}_0\#$ .. index# -1).

LPi.ii cop# $\in$ {<=, <}.

Precondition $\text{index}_0\#$ rop# SUCC(final#).

Invariant $\text{index}_0\# \le \text{index}\#$ rop# SUCC(final#) $\wedge$ PERM(array#, $\text{array}_0\#$) $\wedge$ array#[index#] $\le$ MIN(array# [ind], ind = $\text{index}_0\#$ .. index# -1).

Figure 3 Example of a low-level plan.

Based on the value that instantiates cop#, LPi is specialized to either LPi.i or LPi.ii. LPi is specialized to LPi.i (bubble down the maximum element) when cop# is instantiated to '>' or '>=', and it is specialized to LPi.ii (bubble down the minimum element) if cop# is instantiated to '<' or '<='. Thus, we have a documentation corresponding to the specialization LPi.i and another one corresponding to the specialization LPi.ii. This is given in the documentation field of LPi. For each specialization, the precondition and the invariant necessary for documentation are given.

In the documentation field of LPi, SUCC and PRED denote successor and predecessor functions respectively, PERM(array#, $\text{array}_0\#$) [28, Ch. 20] denotes that array# is a permutation of $\text{array}_0\#$, and MAX(array# [ind], ind = $\text{index}_0\#$ .. index# -1) gives the maximum element of the array having indices between $\text{index}_0\#$ and index $-$ 1 (and similarly MIN).

The precondition and the invariant assert that if $\text{index}_0\#$ rop# SUCC(final#) is true when the loop starts, then $\text{index}_0\# \le$ index# rop# SUCC(final#) remains true through successive iterations of the loop. The invariant of LPi.i also asserts that array# is always a permutation of its initial value and that array#[index#] is always greater than or equal to the maximum of the elements of array# having indices between $\text{index}_0\#$ and index# $-$ 1. The documentation field of LPi.ii can be described similarly.

When the precondition is not satisfied, the while loop does not execute at all. In addition, the while loop does not execute when the initial value of the control variable, $\text{index}_0\#$, equals final# (or final# + 1) and the relational operator, rop#, is instantiated with '< ' (or '$\le$'). In such boundary cases, the precondition is true. However, the first clause of the invariant implies that index# only assumes its initial value $\text{index}_0\#$. Thus, the range in the third clause of the invariant, $\text{index}_0\#$ .. index# $-$ 1, is empty and the invariant is effectively stating that nothing is performed by the while loop.

Figure 4 shows a slice that can be recognized by LPi. The attributes array#, index#, rop#, final# and cop# will be instantiated to 'room_array', 'j', '<=', 'max_rooms - i' and '>' respectively. We will use the documentation of the specialization LPi.i since cop# is instantiated to '>'. The precondition and invariant will be written for the slice after being instantiated with the values of the attributes. Thus, the slice has the following documentation.

Precondition:

$(1 \leq \text{max\_rooms} - i + 1)$.

Invariant:

$(1 \leq j \leq \text{max\_rooms} - i + 1) \wedge$ PERM(room_array, room_array$_0$) $\wedge$ room_array[j] $\geq$ MAX(room_array[ind], ind $= 1 .. j - 1$).

```
j:=1;
while ( j <= max_rooms – i) do begin
      if room_array[j] > room_array[j+1] then begin
            temp := room_array[j];
            room_array[j] := room_array[j+1];
            room_array[j+1] := temp
      end
      j := j+1
end;
```

Figure 4 An example slice recognized by plan LPi.i.

High-Level Plans

High-level plans have the same generic template structure shown in Figure 2. The components are composed of high-level and/or low-level plans depending on the plan's position in the hierarchy. Constraints are on the sharing of information between components. A high-level plan instance is recognized, when all of its component plans are recognized without violating the constraints. Its attributes are then formed from the attributes of its components.

Figure 5 shows a high-level plan HPi. HPi is a high-level plan that can recognize a slice performing an ascending bubble sort. HPi is inferred from two low-level plans LPi.i and LPii.i. It should be mentioned that LPii is a plan for recognizing two nested loops. It has specializations correspond to the four different combinations of enumerations that can take place in two nested loops. LPii has the following attributes.

index2#, index1#: the index variables of the outer and inner loops, respectively.

init2#, init1#: the initial values of the outer and inner loops, respectively.

final2#, final1#: the final values of the outer and inner loops, respectively.

rop2#, rop1#: the relational operators of the outer and inner loops, respectively.

op2#, op1#: the operators used to increment (decrement) index2#, index1#, respectively.

HPi Constraints are on the sharing of information between the two component plans. Some attributes of the two component plans should be instantiated to the same values. An instance of HPi is created when instances of LPi.i and LPii.i are recognized and the constraints binding their attributes are satisfied. In the documentation field of HPi, down_shift(finali#) equals the identity function if ropi# is instantiated to '≤'. Otherwise, it equals the PRED function. The first clauses of the precondition and invariant of HPi assert that if index2$_0$# rop2# SUCC(final2#) is true when the loop starts, then index2$_0$# $\leq$ index2# rop2# SUCC(final2#) remains true through successive iterations of the outer loop. Whereas, the second clauses are added to relate the proofs of both the inner and outer loops. More specifically, they enable the proof of '$I_0 \wedge B_0 \Rightarrow P_i$' for all iterations of the outer loop, where I_0, B_0, and P_i are the outer loop invariant, the outer loop control predicate and the inner loop precondition, respectively. The third clause of the invariant asserts that array# is a permutation of its initial value. The last clause of the invariant ensures that array# is sorted in an ascending order.

Plan name	Hpi
Attributes	array#, index2#, init2#, rop2#, final2#, index1#, init1#, rop1#, final1#.
Components	1. LPi.i(array#, index#, rop#, final#, cop#).
	2. Lpii.i(index2#, init2#, rop2#, op2#, final2#, index1#, init1#, rop1#, op1#, final1#).
Constraints	1. index# = index1#.
	2. rop# = rop1#.
	3. final# = final1#.
Documentation	
Precondition	$init2\#\ rop2\#\ SUCC(final2\#) \wedge$
	$(\forall\ ind: init2\# \le ind \le down\_shift(final2\#) : (PRED(init1\#)\ rop1\#\ final1\#)\vert_{ind}^{index2\#}))$
Invariant	$init2\# \le index2\#\ rop2\#\ SUCC(final2\#) \wedge$
	$(\forall ind: init2\# \le ind \le down\_shift(final2\#) : (PRED(init1\#)\ rop1\#\ final1\#)\vert_{ind}^{index2\#})) \wedge$
	$PERM(array\#, array_0\#) \wedge$
	$(\forall ind: init2\# \le ind \le index2\# - 1: array\#[down\_shift(final1\#)\vert_{ind}^{index2\#} + 1] \ge$
	$MAX(array\#[ind], ind = (init1\#..\ down\_shift(final1\#))\vert_{ind}^{index2\#}).$

Figure 5 Example of a high-level plan.

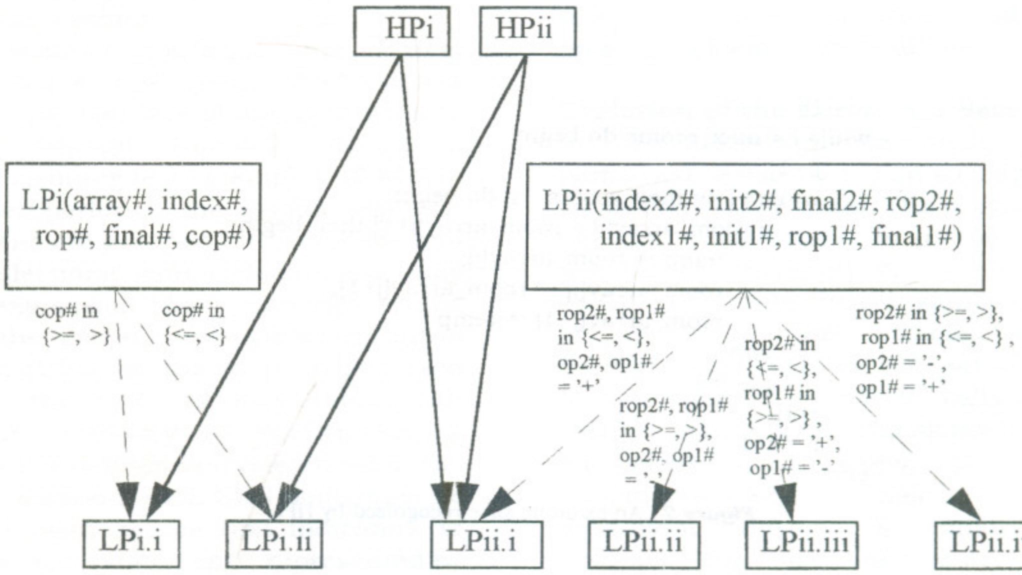
HPi
HPii
LPi(array#, index#, rop#, final#, cop#)
LPii(index2#, init2#, final2#, rop2#, index1#, init1#, rop1#, final1#)
cop# in {>=, >}
cop# in {<=, <}
rop2#, rop1# in {<=, <}, op2#, op1# = '+'
rop2#, rop1# in {>=, >}, op2#, op1# = '-'
rop2# in {<=, <}, rop1# in {>=, >}, op2# = '+', op1# = '-'
rop2# in {>=, >}, rop1# in {<=, <}, op2# = '-', op1# = '+'
LPi.i
LPi.ii
LPii.i
LPii.ii
LPii.iii
LPii.iv

Figure 6 An example to demonstrate our hierarchical knowledge base structure.

HPi fits in our hierarchical knowledge base structure in the way shown in Figure 6. In this figure, only two high-level plans are shown, along with the two low-level plans. The high-level plan HPii is a plan that can recognize a slice performing a descending bubble sort. The solid arrows show how high-level plans are composed of low-level plans. The dashed arrows show the specializations of the low-level plans. High-level plans can also be designed as compositions of low-level and high-level plans.

HPi can be used to recognize the example slice given in Figure 7. Because this example slice is a composite one that consists of two simple slices, it is recognized by a high-level plan, HPi, which consists of two low-level plans. The first step, in the recognition process, is to recognize instances of low-level plans in the slice. Instances of LPi.i and LPii.i are recognized and their attributes are instantiated as described in the previous subssection. Thus, the inner loop is recognized using LPi.i. The two nested loops (without the inner loop body) are recognized using LPii.i. Then, these two identified plans are composed to conform to the hierarchical structure shown in Figure 6. The constraints binding their attributes are checked. Since these constraints are satisfied, an instance of HPi is recognized. HPi's attributes will be filled in as follows: array#, index2#, init2#, rop2#, final2#, index1#, init1#, rop1# and final1# are instantiated to 'room_array', 'i', '1', '<', 'max_rooms', 'j', '1', '<='and 'max_rooms - i' respectively.

The slice is documented, using the documentation field of HPi (given in Figure 5), as follows.

Precondition:
$(1 < \text{max\_rooms} + 1) \wedge (\forall\ \text{ind} : 1 \leq \text{ind} \leq \text{max\_rooms} - 1 : 0 \leq \text{max\_rooms} - \text{ind})$.
Invariant:
$(1 \leq i < \text{max\_rooms} + 1) \wedge (\forall\ \text{ind} : 1 \leq \text{ind} \leq \text{max\_rooms} - 1 : 0 \leq \text{max\_rooms} - \text{ind}) \wedge$
$\text{PERM}(\text{room\_array}, \text{room\_array}_0) \wedge$
$(\forall \text{ind} : 1 \leq \text{ind} \leq i - 1 : \text{room\_array}[\text{max\_rooms} - \text{ind} + 1] \geq \text{MAX}(\text{room\_array}[\text{ind}],\ \text{ind} = 1..\ \text{max\_rooms} - \text{ind}))$.

```
i := 1;
while i < max_rooms do begin
      j:=1;
      while ( j <= max_rooms – i) do begin
            if room_array[j] > room_array[j+1] then begin
                  temp := room_array[j];
                  room_array[j] := room_array[j+1];
                  room_array[j+1] := temp
            end
            j := j+1
      end
      i := i+1
end;
```

Figure 7 An example slice recognized by HPi.

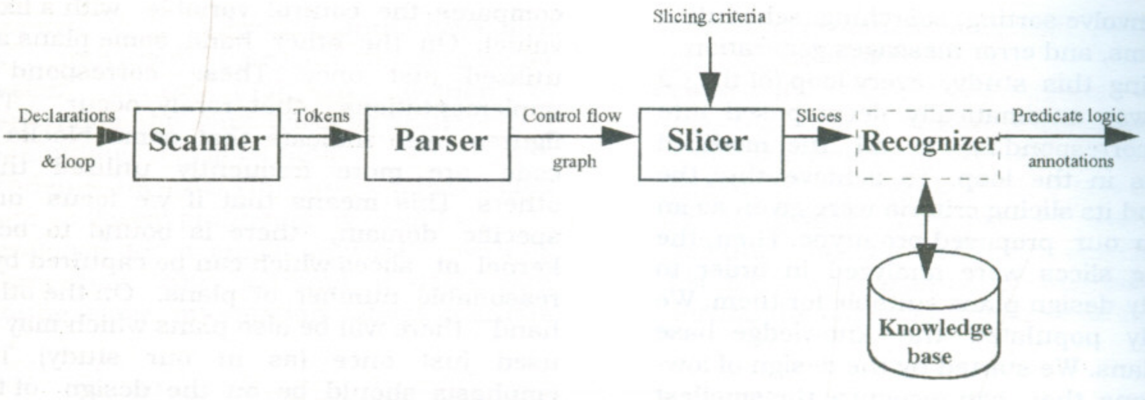

Figure 8 The prototype structure.

IMPLEMENTATION

In this section, we briefly describe a prototype tool that uses program slicing to analyze loops. Our tool automatically decomposes a loop into slices. The resulting slices are then analyzed, using plans stored in a knowledge base, to deduce their predicate logic annotations. Figure 8 depicts the structure of our prototype. The input to the current version is in the form of a loop to be analyzed and its declarations written in a subset of Pascal. It is assumed that the input Pascal program has been previously compiled successfully.

The scanner performs the lexical analysis, where the stream of characters making up the loop is read and grouped into individual strings (tokens) having a collective meaning (e.g., identifier, if, while). The parser performs the syntax analysis, where tokens are grouped hierarchically into nested collections with collective meaning (e.g., *if* expression *then*). Through syntax analysis, the parser constructs the control flow graph (CFG) of the loop, where each node in the CFG is augmented with information necessary to perform slicing (e.g., variables defined and referenced at the node). In addition, each CFG node must be augmented with source text indicators to the actual text (statement) represented by this CFG node. The lexical and syntax analysis are completely automated. The slicer implements Weiser's slicing algorithm. Slicing criteria are supplied by the user. Slices are produced automatically for each supplied criterion. The recognizer matches the resulting slices against the knowledge base plans and produces predicate logic annotations. In the current version, the tasks of the recognizer are performed manually.

EVALUATION

In this section, we test and evaluate our proposed analysis approach. First, we describe how we created the knowledge base, to get an initial set of plans. Then, we show how this initial set was utilized to recognize loops in real and existing programs of some practical use.

Evolution of the Knowledge Base

To create our knowledge base, we performed a case study on a complete set of loops in a real program. The program chosen for the case study is in the domain of scheduling university courses [29]. It has 1,400 executable Pascal source code. A set of 62 loops are extracted with their initializations. This is the set of loops that do not contain procedure calls. All *for* and *repeat* loops were transformed to their equivalent *while* loops, and all *case* statements were transformed to their equivalent nested *if* constructs. The analyzed loops are single entry/single exit structured loops, which have the usual programming language features such as pointers, arrays, records, functions with no side effects and nested loops. Many of these loops involve sorting, searching, scheduling algorithms, and error messages generation.

During this study, every loop (of the 62 loops) was automatically decomposed into slices corresponding to all the modified variables in the loop. To achieve this, the loop and its slicing criteria were given as an input to our proposed prototype. Then, the resulting slices were analyzed in order to manually design plans suitable for them. We gradually populated the knowledge base with plans. We started by the design of low-level plans that can recognize the smallest slices. These low-level plans were designed to be general enough, to be used in recognizing as much slices as possible. For slices having larger sizes, we composed low-level and/or high-level plans to create new high-level plans. Some of the designed low-level plans did not correspond to syntactically reasonable section of code. They were designed only to show how existing low-level/high-level plans fit together, and how they are related to one another (i.e., to show the ordering between instances of the recognized plans). After analyzing all the slices, we got the previously described hierarchical knowledge base structure. Table 2 shows the total numbers of slices, low-level plans (LPs), and high-level plans (HPs) resulting from applying our analysis approach on the 62 loops.

The dark bars in Figures 9 and 10 show the utilization of the low-level and high-level plans during the analysis of the 196 slices. The low-level plans were used 296 times (78.7%), whereas the high-level plans were used 80 times (21.3%).The average and standard deviation of the number of utilizations of the low-level plans are 7 and 8, respectively. The average and standard deviation of the high-level plans are 3.3 and 3.4, respectively. We notice that, the plan utilizations are skewed towards the low-level plan LP1. This low-level plan is used to match a simple slice, that just iterates from an initial value to a final value. If we perform slicing on the loop's control variable, this is a common slice in almost all loops having a single clause control condition (that compares the control variable with a fixed value). On the other hand, some plans are utilized just once. These correspond to implementations that rarely occur. The figures also indicate that some blocks of code are more frequently utilized than others. This means that if we focus on a specific domain, there is bound to be a kernel of slices which can be captured by a reasonable number of plans. On the other hand , there will be also plans which may be used just once (as in our study). The emphasis should be on the design of the plans that cover the kernel.

Utilization of the Knowledge Base

In this section, we investigate the effect of using the existing knowledge base, designed for the scheduling program, to recognize slices from other programs. Five programs of varying characteristics were chosen from a recent publications [30], where the author randomly chose these programs for evaluation of a knowledge-based approach that is based on decomposition. A set of 70 loops were extracted from these programs. The initial step for this study, is to apply slicing on the 70 loops. Then, we examine the utilization of our existing knowledge base and its ability to recognize the new slices.

Table 2 Statistical data for the university scheduling program.

# of loops	# of slices	# of LPs	# of HPs	Total # of plans
62	196	42	24	66

To provide some insight into the differences between the five programs, will give a brief description of each of them.

- Set of general purpose loops: These are loops included in the programming examples of a Pascal user manual [31].
- Scanner 1: This is a Pascal scanner designed in an advanced programming book [32]. It converts from characters representing a number to integer, processes single or double character special strings, and recognizes literal characters surrounded by quotes.

- Scanner 2: It is Pascal scanner developed by a university instructor for a compiler design course [33]. In addition to performing the same tasks as Scanner 1, it manipulates a symbol table.
- Students records: This program is selected from a Pascal programming book [34]. It is developed for a small university with no more than 200 students. It deals with storing some information about graduate and undergraduate students, sorting and printing this information.
- Linked list system: This program , which is selected form a Pascal programming book [34], deals with the development of a small linked list processing system. It is a menu-driven program that permits inserting or deleting nodes from the list, searching the list for an occurrence of some value, and sorting the list according to some value.

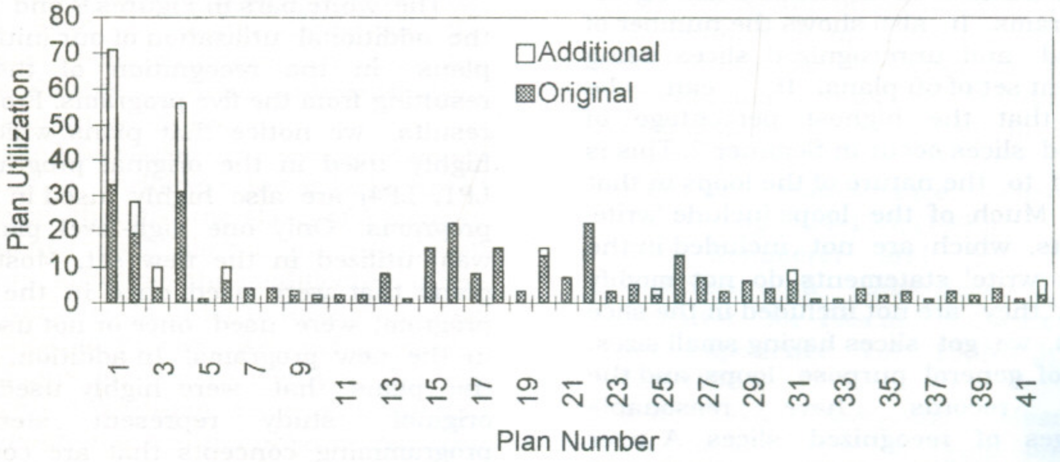

Figure 9 Original and additional utilization of the low-level plans.

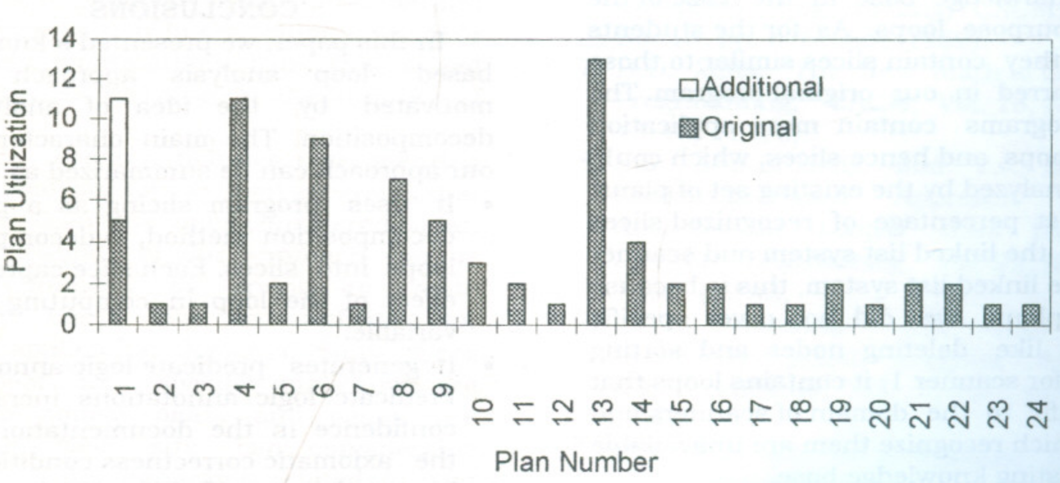

Figure 10 Original and additional utilization of the high-level plans.

Table 3 shows the number of loops extracted and the number of executable lines of code for each of the five programs. We have extracted a set of 70 loops from a total of 92 loops from the five programs. This is the set of all loops within these five programs that do not contain procedure calls. All *for* and *repeat* loops were transformed to their equivalent while loops, and all *case* statements were transformed to their equivalent nested *if* constructs. The analyzed loops are single entry/single exit structured loops, which have the usual programming language features such as pointers, arrays, records, functions with no side effects and nested loops.

We applied slicing on every loop in the set of 70 loops on every modified variable. We got a set of 161 slices. Table 4 shows how this number is distributed among the five programs. It also shows the number of recognized and unrecognized slices using our current set of 66 plans. It can be noticed that the highest percentage of recognized slices occur in Scanner 2. This is attributed to the nature of the loops in that program. Much of the loops include 'write' statements, which are not included in the slices — 'write' statements do not modify variables, they are not included in the slice and thus, we got slices having small sizes. The set of general purpose loops, and the students records have reasonable percentages of recognized slices. A user manual contains many stereotyped programming concepts which were captured by our knowledge base in the case of the general purpose loops. As for the students records, they contain slices similar to those that occurred in our original program. The other programs contain more application specific loops, and hence slices, which could not be analyzed by the existing set of plans. The lowest percentage of recognized slices occur in the linked list system and scanner 1. For the linked list system, this is because in our plans, we did not cover specific concepts like deleting nodes and sorting lists. As for scanner 1, it contains loops that are specific to the domain of scanners and plans which recognize them are unavailable in the existing knowledge base.

Table 3 Statistical data of the five programs.

Program	# of lines	# of loops
Set of general purpose loops	—	30
Scanner 1	483	12
Scanner 2	654	12
Student records	236	10
Linked list system	277	6
Total	—	70

Table 4 Number of recognized and unrecognized slices.

Program	# of slices	# of recogniz ed slices	# of unrecognized slices
Set of general purpose loops	91	53 (58.3%)	38 (41.7%)
Scanner 1	20	9 (45%)	11 (55%)
Scanner 2	18	12 (66.7%)	6 (33.3%)
Student records	20	11 (55%)	9 (45%)
Linked list system	12	5 (41.6%)	7 (58.3%)
Total	161	90 (55.9%)	71 (44.1%)

The white bars in Figures 9 and 10 show the additional utilization of our initial set of plans in the recognition of the slices resulting from the five programs. From these results, we notice that plans which were highly used in the original program (e.g., LP1, LP4) are also highly used in the new programs. Only one high-level plan (HP1) was utilized in the new set. Most of the plans that were used once in the original program, were used once or not used at all in the new programs. In addition, many of the plans that were highly used in the original study represent stereotyped programming concepts that are commonly used across different applications.

CONCLUSIONS

In this paper, we presented a knowledge-based loop analysis approach. It is motivated by the idea of analysis by decomposition. The main characteristics of our approach can be summarized as follows:

- It uses program slicing as a practical decomposition method, to decompose the loops into slices. Each slice captures the effect of the loop in computing a single variable.
- It generates predicate logic annotations. Predicate logic annotations increase the confidence in the documentation, since the axiomatic correctness conditions can be stated and verified if desired .
- It is a knowledge-based approach, that builds a hierarchical structure of plans.

We developed a prototype tool that implements the decomposition method. A hierarchical knowledge base having a set of 66 plans was also designed. In the current implementation, generation of the predicate logic annotations is performed manually.

Finally, an evaluation of our proposed approach was performed. This evaluation serves to show the effect of using program slicing, and our plan design method on the acquisition and development of plans in the knowledge base. By using an initially designed set of plans to analyze five different programs, we demonstrated that the knowledge base generated for a given program is generally usable beyond that program. That is, using slicing and the hierarchical plan structure can make the plans applicable in many different slices across different applications.

Yet, the knowledge base size is relatively large. This is attributed to the large size of slices. Even though the slices of a loop are independent, each slice must include all statements affecting the modification of the current variable. In order to have plans that are reasonable in size, we had to design several plans for the recognition of every large slice. In addition, some loops specific to the domain of the initial program required the design of many plans to recognize their slices (e.g., one loop required the design of 6 plans). These plans were not utilized beyond those slices, either in the initial program or in the five used programs. Thus, this caused a considerable increase in the number of plans.

Future research includes experimenting with the analysis approach on programs that are larger than the ones considered so far. The practicability of our approach can be greatly enhanced by trying to create knowledge bases that are sufficient for specific application domains. Since slices that appear in a specific application are likely to appear again and again, the knowledge base size can considerably decrease. Thus, programs related to that application domain can be documented in an efficient method. Finally, our current prototype tool needs to be enhanced to support additional programming language features such as procedure calls. In addition, more work is required for the automation of all the analysis steps, and to make the interface more user friendly.

REFERENCES

1. E. Soloway, J. Bonar, K. Ehrlich, "Cognitive Strategies and Looping Constructs: an Empirical Study," CACM, Vol. 26, No. 11, pp. 853-861, (1983).
2 S.K. Abd-El-Hafiz, V.R. Basili and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," Proc. of Conf. on Software Maintenance, Sorrento, Italy, pp. 212-219, (1991).
3. A. Cimitile, A. De Lucia and M. Munro, "Identifying Reusable Fnctions Using Specification Driven Program Slicing: a Case Study," Proc. of Int'l Conf. on Software Maintenance, Opio (Nice), France, pp. 124-133, (1995).
4. R.A. Kemmerer and S.T. Eckmann, "UNISEX: a Unix-Based Symbolic Executor for Pascal," Software Practice and Experience, Vol. 15, No. 3, pp. 439-458, (1985).
5 R.W. Floyd, "Assigning Meanings to Programs," Proc. of Symp. on Applied Mathematics, pp. 19-32, (1967).
6. C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," CACM, Vol. 12, No. 10, pp. 576-580, (1969).
7. H.D. Mills, "The New Math of Computer Programming," CACM, Vol. 18, No. 1, pp. 43-48, (1975).
8. S.K. Abd-El-Hafiz, and V.R. Basili, "A Knowledge-Based Approach to the Analysis of Loops," IEEE Trans. on Software Engineering, Vol. 22, No. 5, pp. 1-22, (1996).
9. M.T. Harandi, and J.Q. Ning, "Knowledge-Based Program Analysis," IEEE Software, Vol. 7, No., pp. 74-81, (1990).
10. W.L. Johnson and E. Soloway, "PROUST: Knowledge-Based Program Understanding", IEEE Trans. on Software Engineering, Vol. 11, No. 3, pp. 267-275, (1985).
11. A. Quilici, "A Memory Based Approach to Recognizing Programming Plans," CACM, Vol. 37, No. 5, pp. 84-93, (1994).

12 L.M. Wills, "Flexible Control for Program Recognition," Proc. of 1st Working Conf. on Reverse Engineering, Baltimore, Maryland, pp. 134-143, (1993).

13.S.K. Abd-El-Hafiz and V.R. Basili, "A Knowledge-Based Approach to Program Understanding," Kluwer Academic Publishers, (1995).

14.J. Hartman, "Understanding Natural Programs Using Proper Decomposition," Proc. of 13th Int'l Conf. on Software Engineering, Austin, Texas, pp. 62-73, (1991).

15.P.A. Hausler, M.G. Pleszkoch, R. Linger, and A.R. Hevner, "Using Function Abstraction to Understand Program Behavior," IEEE Software, Vol. 7, No. 1, pp. 55-63, (1990).

16.M. Weiser, "Program slicing", IEEE Trans. on Software Engineering, Vol. 5, No. 3, pp. 237-247, (1979).

17.S. Horwitz, T. Reps and D. Binkly, "Interprocedural Slicing Using Dependence Graphs," ACM TOPLAS, Vol.12, No.1, pp. 26-40, (1990).

18 B. Korel and J. Laski, "Dynamic Program Slicing", Information processing letters, Vol. 29, pp. 155-163, (1988).

19.K.J. Ottenstein and L.M. Ottenstein, "The Program Dependence Graph in a Software Development Environment," ACM SIGPLAN Not., pp. 177-184, Vol. 19, No.5 (1984).

20.M. Weiser, "Programmers Use Slices When Debugging," CACM, Vol. 25, No. 7, pp. 446-452, (1982).

21.Lyle, J.R. and Gallagher, K.B., "Using Program Decomposition to Guide Modifications, " Proc. of Conf. on Software Maintenance, pp. 265-268, (1988).

22.Gallagher, K.B. and Lyle, J.R., "Using program slicing in software maintenance," IEEE Trans. on Software Engineering, Vol. 17 , No. 8, pp. 751-761, (1991).

23.J. Jiang, X. Zhou and D. Robson, "Program Slicing for C - the Problems in Implementation", Proc. of Conf. on Software Maintenance, Sorrento, Italy, pp. 212-219, (1991).

24.M.A. Sorour, "Program Slicing in Knowledge-Based Loop Analysis," M.Sc. Thesis, Faculty of Engineering, Cairo University, Giza, Egypt, (1997).

25.F. Tip, "A Survey of Program Slicing Techniques," Journal of Programming Languages, Vol. 3, pp. 121-189, (1995).

26.H. Leung and H. Reghbati, "Comments on Program Slicing", IEEE Trans. on Software Engineering, Vol. 13, No. 12, pp. 1370-1371, 1987.

27.S. Woods, Q. Yang, "Program Understanding as Constraint Satisfaction, "Proc. of 2nd working Conf. on Reverse Engineering, Toronto, Ontario, Canada, pp. 314-323, (1995).

28.D. Gries, "The Science of Programming," Springer-Verlag, (1981).

29.P. Jalote, "An Integrated Approach to Software Engineering," Springer Verlag, (1991).

30.S.K. Abd-El-Hafiz, "Evaluation of a Knowledge-Based Approach to Program Understanding," Proc. of Int'l Conf. on Software Maintenance, Monterey, California, pp. 275-284, (1996).

31.K. Jensen and N. Wirth, "Pascal User Manual and Report", Springer-Verlag, (1974).

32.Schneider, and Bruell, "Advanced Programming and Problem Solving with Pascal," John Wiley, (1987).

33.C.W. Barth, "Table-Driven Scanner," the computer project for CMSC 430 course offered at the University of Maryland, College Park, MD, (1985).

34. E.L. Lamie, "Pascal Programming", John Wiley, (1987).

Received February 6, 1998
Accepted May 9, 1998

تحليل العروات باستخدام قواعد المعلومات عن طريق تجزئة البرامج

مروة عارف سرور، سلوى كمال عبد الحفيظ

قسم الرياضيات والفيزياء الهندسية - جامعة القاهرة

ملخص البحث

تؤثر العروات المستخدمة فى البرامج على فهمها و ذلك للصعوبات المرتبطة بها ، يناقش هذا البحث طريقة آلية لفهم عروات البرامج باستخدام قواعد المعلومات. فى هذه الطريقة يتم تقسيم العروة باستخدام طريقة معروفة لتجزئة البرامج الى شرائح عن طريق تحليل التحكم و تدفق البيانات، و تعزل كل شريحة تأثير العروة فى حساب متغير واحد فقط، ثم يتم تحليل الشرائح الناتجة باستخدام نماذج تسمى "خطط" بقاعدة معلومات لتوثيقها باستخدام المنطق التنبؤى من الدرجة الأولى **"First-Order Predicate Logic"** . و نظرا للأساس الرياضى القوى لهذا التوثيق، يمكن اثبات صحته باستخدام المنهج التصحيحى المسلمى **"Axiomatic Correctness Approach"** . و لتقييم هذه الطريقة تم استخدامها فى تحليل مجموعة من العروات حيث تم تجزئتهم الى شرائح ثم صممت مجموعة من الخطط قادرة على تحليل و توثيق هذه الشرائح. و بذلك تم بناء قاعدة معلومات أولية يمكن استخدامها فى فهم شرائح من برامج أخرى. و تم تطبيق الطريقة على فهم مجموعة شرائح من برامج أخرى ذات حجم مناسب و قيمة عملية، و أظهرت النتائج امكانية استخدامها.