

Evaluation of a Knowledge-Based Approach to Program Understanding

Salwa K. Abd-El-Hafiz
Engineering Mathematics Department
Faculty of Engineering, Cairo University
Giza, Egypt.

Abstract

This paper evaluates a recently presented knowledge-based program understanding approach that generates first order predicate logic annotations of loops. An initial and partial evaluation of this approach was performed on an existing program of reasonable size. Plans in the knowledge base were designed by performing an analysis of the existing program. As a result, the previous study did not demonstrate that the knowledge base generated for a given program is generally usable beyond that program. In this study, an extensive evaluation of the analysis approach is performed. Plans developed during the initial evaluation stage are used to analyze a set of 92 loops in five randomly selected Pascal programs. Results concerning the analyzed loops and utilization of the existing plans are given and discussed. These results generally show a good usability of the knowledge base beyond the original program.

1. Introduction

Recently, we presented a knowledge-based approach to the automation of program analysis [1, 2, 3, 4, 7]. It combines and builds on the strengths of a practical program decomposition method [22], the axiomatic correctness notation [11], and the knowledge-based analysis approaches [9, 10, 13, 15, 18, 19]. It mechanically documents programs by generating first order predicate logic annotations of their loops. This approach can assist in the maintenance activities by producing semantically sound and expressive predicate logic annotations of programs. Since many programs are undocumented, underdocumented, or misdocumented, a major part of the maintenance task is spent in recognizing and understanding abstract programming concepts. Automation of program analysis and understanding can, thus, contribute to maintenance tools and methods and provide support for various maintenance activities.

In this approach, a family of analysis techniques has been developed and tailored to cover different levels of program complexity. This complexity is determined by classifying while loops along three dimensions. Then, we annotate loops with predicate logic assertions in a step by step process as depicted in Figure 1 [1, 2]. The analysis of a loop starts by decomposing it into fragments, called *events*. Each event encapsulates the loop parts that are interdependent, with respect to data flow, and separates them from the rest of the loop. The resulting events are then analyzed, using plans stored in a knowledge base, to deduce their individual predicate logic annotations. Finally, the annotation of the whole loop is synthesized from the annotations of its events.

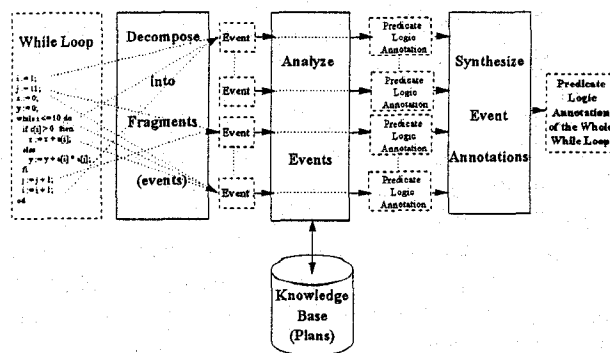


Figure 1. Overview of the analysis approach.

In order to evaluate our analysis approach, we previously performed a case study on a set of 77 loops in an existing Pascal program [12] for scheduling university courses. Analysis results supported the following two hypotheses:

- Loop complexity dimensions are valid indicators of its amenability to analysis.
- The loop decomposition and plan design methods of our approach can make the plans applicable in many different loops and, hence, increase their utilization.

The results of this study served to partially evaluate our analysis approach and to assess its effectiveness when applied to a fixed set of loops in a real and existing program of some practical value [1, 2]. However, the

plans in the knowledge base were designed by performing an analysis of the existing program. As a result, the study did not demonstrate that the knowledge base generated for a given program is generally usable beyond that original program.

This paper describes a new extensive study performed to investigate the usability of the existing knowledge base with respect to different programs. In this study, we randomly chose five programs of varying characteristics from several Pascal sources [6, 14, 21, 23]. Then, we analyzed the set of 92 loops included in these five programs using the same analysis approach and the same set of existing plans. In addition to investigating the usability of the knowledge base, we further validate the aforementioned two analysis hypotheses and test the following hypothesis:

- Many of the plans that were highly used in the original study represent stereotyped programming concepts that are commonly used across different applications.

Section 2 of this paper outlines the knowledge-based analysis approach that we evaluate. Section 3 describes how the current study was performed and discusses its results. Finally, conclusions and future research directions are given in Section 4.

2 The knowledge-based loop analysis approach

Let the *abstract representation of the while loop* be *while B do S* where the condition *B* has no side effects and the statements *S* are representable by a single-entry single-exit control-flow graph. This representation abstracts from the syntax of the specific imperative programming language being used. Though the approach described here applies to all loops having this abstract representation, examples and illustrations are given using Pascal. Using this abstract representation, a *control variable* of the while loop is a variable that exists in the condition *B* and is modified in the body *S*.

A *concurrent assignment* is a statement in which several variables can be assigned simultaneously. We use the form $v_1, v_2, \dots, v_n := e_1, e_2, \dots, e_n$ to assign every *i*th expression from the right hand list to its corresponding *i*th variable from the left hand list [8, 16]. A *conditional assignment* is a set of one or more guarded concurrent assignments separated by commas ','. When the guard (i.e., the boolean expression) of a concurrent assignment is satisfied, the modifications performed on a variable are given by the concurrent assignment [8, 16]. All the guards must be well defined [8]. However, it is possible that all guards evaluate to false. In this case, no variable is modified. It should also be noted that because we are

only analyzing deterministic programs, all the guards are mutually exclusive.

2.1 Loop taxonomy

To design the analysis techniques that best fit different levels of program complexity, we classify while loops along three dimensions. The first dimension focuses on the control computation part of the loop. The other two dimensions focus on the complexity of the loop condition and body. Along each dimension, a loop must belong to one of two complementary classes as shown in Table 1. In this classification, loops in the middle column are expected to be more amenable to analysis than the corresponding ones in the right column.

Within the first dimension, we differentiate between simple and general loops. *Simple loops* have a behavior similar to that of *for* loops. They are defined by imposing two restrictions: the loop has a unique control variable, and the modification of the control variable does not depend on the values of other variables modified within the loop body. Loops that do not satisfy these conditions are called *general loops*.

Along the second dimension, the complexity of the loop condition varies between two cases. In the *noncomposite* case, *B* is a logical expression that consists of one clause of the conjunctive normal form [20]. In the *composite* case, more than one clause exists. Along the third dimension, the complexity of the loop body varies between *flat* and *nested* loop structures. In flat loop structures, the loop body can not contain any other loop inside it which is not the case in nested structures.

Table 1. The three dimensions used for classifying loops.

Dimension	Complementary classes	
	1. Control computation	Simple loop
2. Complexity of condition	Noncomposite condition	Composite condition
3. Complexity of body	Flat loop	Nested loop

2.2 Loop decomposition

A loop is decomposed into fragments, called *events*. Each event encapsulates the loop parts that are interdependent, with respect to data flow, and separates them from the rest of the loop. The three main phases of the decomposition are normalization of the loop representation, decomposition of the loop body, and formation of the loop events. Descriptions of these

phases [4] and their application to the example shown in Figure 2 are given in the remainder of this section. In this example, a simple loop with a noncomposite condition scans a segment of the array *capacity* searching for its minimum.

```

j, index, min, num_of_rooms: integer;
capacity: array[1 .. max_rooms] of integer;
:
index := i;
min := capacity[i];
j := i + 1;
while j < num_of_rooms + 1 do begin
  if capacity[j] < min then begin
    index := j;
    min := capacity[j];
  end;
  j := j + 1
end;

```

Figure 2. Example loop.

2.2.1 Normalization of the loop representation. The purpose of this phase is to make the loop representation independent of the programming language and the implementation specific details. The loop condition is converted into a standard normal form, which is the *conjunctive normal form* [20]. For example, the loop condition $x < a$ or $(y < b$ and $z < c)$ is transformed to a conjunction of two clauses. The first clause is $(x < a$ or $y < b)$ and the second is $(x < a$ or $z < c)$. A single unwinding of the loop body is performed by symbolic execution [5] that gives the net modification performed on each variable in one iteration of the loop, if any [7]. We use the conditional assignment notation to represent the result of this symbolic execution. After converting the loop condition and body into the aforementioned standard forms, they are further normalized by performing some arithmetic and predicate simplifications [17].

For the loop given in Figure 2, the condition is already in conjunctive normal form containing the one clause $j < \text{num_of_rooms} + 1$. The symbolic execution does not change the body of the loop. However, the net modification performed on each variable is given in the form of a conditional assignment as follows:

```

capacity[j] < min  $\Rightarrow$  index := j,
capacity[j] < min  $\Rightarrow$  min := capacity[j],
true  $\Rightarrow$  j := j + 1

```

2.2.2 Decomposition of the loop body. To facilitate the mechanical generation of loop annotations, the symbolic execution result is uniquely decomposed, using data flow analysis, into *segments* of code that can be analyzed

separately. These *loop segments* are partitions of the loop body symbolic execution result. Each segment consists of a maximal set of conditional assignments such that any two conditional assignments in the set are circularly dependent (i.e., each segment is responsible for the data flow into the other one) [1, 2]. Because the analysis results of a segment might affect the analysis of other segments dependent on it, the segments are ordered according to their data flow dependencies [22]. The detailed algorithms for forming the loop segments and ordering them are described elsewhere [1, 2]. For the loop shown in Figure 2, the ordered segments of the body are:

Order	Segment
1	$\text{true} \Rightarrow j := j + 1$
2	$\text{capacity}[j] < \text{min} \Rightarrow \text{min} := \text{capacity}[j]$
3	$\text{capacity}[j] < \text{min} \Rightarrow \text{index} := j$

Notice that the segment that defines j has the lowest order because the other two segments reference j . Similarly, the reason for ordering the last two segments is that min is defined in the second segment and referenced in the third one.

2.2.3 Formation of the loop events. To represent the abstract concepts in a loop, we use the loop body segments and the clauses of the loop condition to form the *loop events*. We define two categories of loop events: *basic events* and *augmentation events*.

Basic Events (BEs) are the fragments that constitute the control computation of the loop. A BE consists of three parts: the *condition*, the *enumeration*, and the *initialization*. The *condition* consists of only one clause from the loop condition. The *enumeration* is a segment responsible for the data flow into the *condition* (i.e., the variables assigned in the *enumeration* are referenced by the *condition*). The *initialization* is the initialization of the variables defined in the *enumeration*.

Augmentation Events (AEs) are the fragments that constitute loop computations other than the control computation. An AE consists of two parts: the *body* and the *initialization*. The *body* is one segment of the loop body that is not responsible for the data flow into the loop condition. The *initialization* is the initialization of the variables defined in the *body*.

To form BEs and AEs, refer to the detailed description in [1, 2]. We give each event (basic or augmentation) the same order as the segment it utilizes. This enforces the condition that the variables referenced in an event are either defined in a lower order event or not modified within the loop at all. As mentioned in the previous subsection, this makes it possible to propagate the results of analyzing an event to the analysis of other events dependent on it.

The three ordered events of the loop shown in Figure 2 are:

1. BE (order 1)
condition: $j < num_of_rooms + 1$
enumeration: $true \Rightarrow j := j + 1$
initialization: $j := i + 1$
2. AE (order 2)
body: $capacity[j] < min \Rightarrow min := capacity[j]$
initialization: $min := capacity[i]$
3. AE (order 3)
body: $capacity[j] < min \Rightarrow index := j$
initialization: $index := i$

2.3 A knowledge base of plans

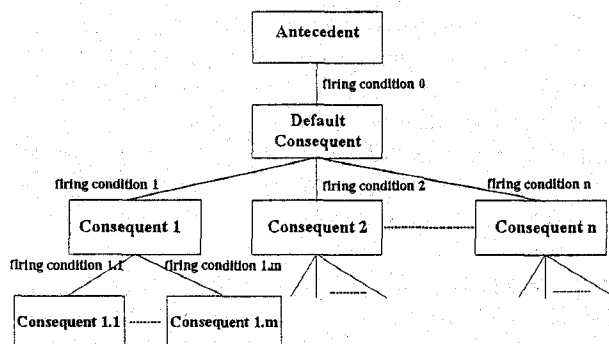


Figure 3. The tree structure of a plan.

To analyze the loop events, tree-structured plans [1, 2, 9, 10, 13, 15, 18, 19] stored in a knowledge base are used. The term 'plan' is used to refer to a unit of knowledge required to identify an abstract concept in a program. A tree-structured plan consists of a single antecedent and several consequents organized into one or more tree structures as shown in Figure 3. The root of the tree corresponds to an antecedent part that should match loop events. The edges of the tree correspond to local firing conditions that control the selection of the appropriate consequents given in the remaining tree nodes. In order to select a specific tree-structured plan, the event under consideration should satisfy the antecedent first. Within the plan, local firing conditions guide the search for the suitable consequent. The more general the consequent, the closer it is to the root of its tree (e.g., consequent 1 of Figure 3 is more general than consequent 1.1). Firing conditions located at the same level are mutually exclusive. This means that only forward search is needed and no backtracking is required.

In general, an antecedent contains generic patterns of BEs and AEs that are used to match stereotyped loop events. A firing condition contains knowledge needed for the correct identification of the plans such as data

type information and the results of analyzing previous events. A consequent includes knowledge necessary for the annotation of loops with their Hoare-style predicate logic specifications such as the precondition, invariant, and postcondition [11].

Corresponding to the two event categories, we have two plan categories: *Basic Plans (BPs)* and *Augmentation Plans (APs)*. BPs analyze BEs and APs analyze AEs. Plans are further classified according to the kind of loops they analyze. To accommodate the differences between simple and general loops, we have two categories of BPs. Namely, *Determinate BPs (DBPs)* and *Indeterminate BPs (IBPs)*. Similarly, we have two categories of APs which are *Simple APs (SAPs)* and *General APs (GAPs)*. These plan categories are shown in the two middle levels of Figure 4.

2.4 Analysis of events and synthesis of their analysis results

When an event satisfies the antecedent of a plan, the appropriate consequent of the matched plan is instantiated giving the contribution of the event to the loop specification. The precondition, invariant and postcondition are formed by taking the conjunction of the corresponding parts of the event analysis results. In the case when some event(s) do not match any knowledge base plans, we get partial specifications of the loop. The synthesized analysis results of loop given in Figure 2 are:

Precondition:

$(i < num_of_rooms + 1)$

Invariant:

$(i + 1 \leq j < num_of_rooms + 2)$ and
 $(min = MIN(capacity[i .. j - 1]))$ and
 $(capacity[index] = min)$

Postcondition:

$(j = num_of_rooms + 1)$ and
 $(min = MIN(capacity[i .. num_of_rooms]))$ and
 $(capacity[index] = min)$

Analysis of nested loops is performed by recursively analyzing the innermost loops and replacing them with sequential constructs, called abstraction classes, that represent their functional abstraction. Using this analysis technique, the difference between the analysis of flat loops and outer loops in nested constructs is that the analysis of outer loops might use high-level plans in addition to the usual (low-level) ones. For a definition of high-level plans, refer to [1, 2]. In addition, a complete proof of the resulting nested loop specifications is enabled by adapting inner loop specifications to the context and initialization provided by outer loops [1, 2].

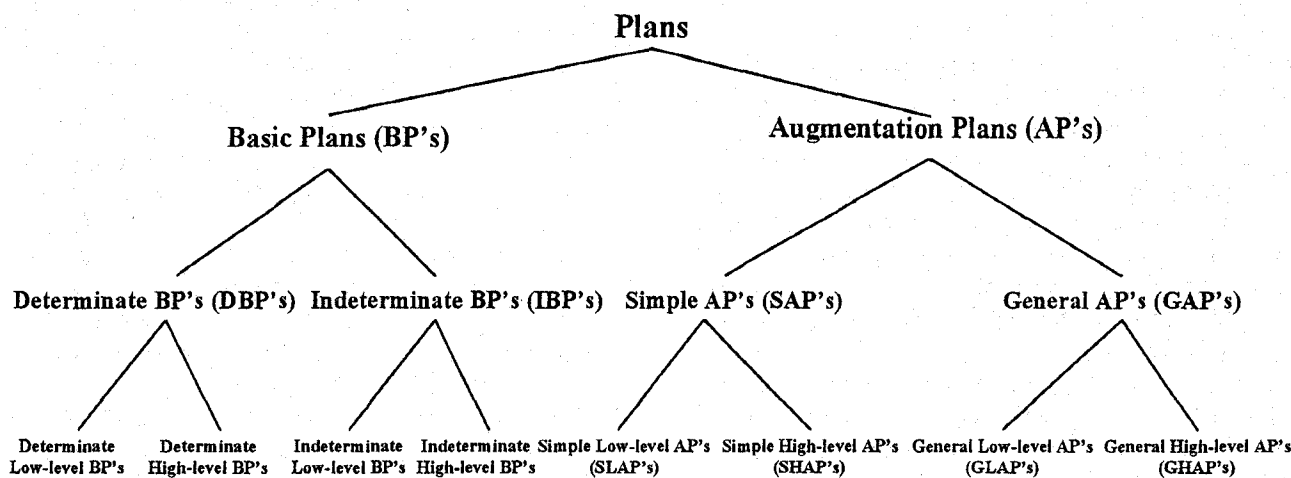


Figure 4. Plan categories.

Adding to the previously defined four plan categories another classification level, based on whether the plan is low-level or high-level, we get 8 plan categories. These new plan categories are shown in the lowest level of Figure 4. The advantage of this plan classification scheme is that it indexes plans for rapid access given the loop and event types.

3 Evaluation

The preliminary evaluation of our analysis approach was previously done by performing a case study on a set of 77 loops in an existing Pascal program [12] for scheduling university courses. These 77 loop included 356 events. In that study, we designed the plans needed for analyzing as many events as possible from the 356 events. Analysis results supported the following two hypotheses:

- Loop complexity dimensions are valid indicators of its amenability to analysis.
- The suggested loop decomposition and plan design methods can make the plans applicable to many different loops and, hence, increase their utilization.

The original case study served to partially validate the analysis hypotheses and to characterize the practical limits of the analysis approach (for the detailed results, refer to [1, 2]). Because the original program was analyzed to construct the plans in the knowledge base, that study did not demonstrate that the knowledge base generated for a given program is generally usable beyond that original program.

The main objectives of the current study are:

- To investigate the usability of the previously developed knowledge base with respect to different programs.

- To further validate the aforementioned two analysis hypotheses.
- To test the hypothesis that many of the plans which were highly used in the original study represent stereotyped programming concepts that are commonly used across different applications.

Five programs of varying characteristics were randomly chosen for these purposes from several Pascal sources [6, 14, 21, 23]. The set of 92 loops included in these five programs were then analyzed using the described analysis approach.

3.1 The analyzed programs

To provide some detailed insight into the differences between the programs analyzed, we give a brief description of each of them.

Original program. This program is for scheduling a set of courses offered by a computer science department [12]. It takes two file inputs. File 1 contains information about the courses in the department catalog, rooms in the building, and valid lecture times. The second file contains, for each course offered in a given semester, the course number, its enrollment and a set of time preferences given by the instructor of the course. The goal is to schedule these courses in the rooms such that a set of constraints given in a requirements document is satisfied. The program contains loops that involve sorting, searching, and scheduling algorithms. Because of the interactive nature of this program, it contains several other loops that perform error detection as well as warning and error messages generation. Loops analyzed have the usual programming language features such as pointers, procedure and function calls, and nested loops. These loops do not involve recursive

function and procedure calls. Recursion is not currently being handled by our analysis approach.

Set of general purpose loops. These are the loops included in the programming examples of a Pascal user manual [23]. Loops included in the chapter about procedures and functions were excluded because most of them were either recursive or similar to loops in preceding chapters. Two small examples, included in the chapter about file types, were also excluded.

Scanner I. This is a Pascal scanner designed in an advanced programming book [21]. It performs the following tasks:

1. Converting from characters representing a number to integer, real, or octal numbers.
2. Collecting together the characters of a non-numeric string.
3. Processing single or double character special strings.
4. Recognizing literal strings surrounded by quotation marks.
5. Detecting and recovering from errors in processing illegal input.

Scanner II. It is a Pascal scanner developed by a university instructor for a compiler design course [6]. It is more powerful than scanner I. In addition to performing the tasks performed by scanner I, it manipulates a symbol table and dumps all internal tables for verification.

Student records. This program is selected from a Pascal programming book [14]. It is developed for a small university with no more than 200 students. The university wants to store some information about its graduate and undergraduate students, then sort and print this information by student name but where graduate students appear before undergraduate students. Information about these students is retrieved on request from the program user. Specifically, the following items are to be retrieved when requested: the number of undergraduate and graduate students, and information for a specific student.

Linked list system. This program, which is selected from a Pascal programming book [14], deals with the development of a small linked list processing system. It is a menu-driven system that permits the creation of up to 10 linked lists. For each of the lists, we can insert nodes, delete nodes, search the list for an occurrence of some value, and sort the list according to some value.

3.2 Method

The set of 92 loops included in the randomly selected five programs were extracted along with their initializations. This set included 29 *for* loops and 11 *repeat* loops, which were transformed to their equivalent

while loops. This transformation was the only modification performed on the input data. As in the original program, loops included in the five programs had the usual programming language features such as pointers, procedure and function calls, and nested loops.

During the study, every loop under consideration was first decomposed into its basic and augmentation events. Then, plans in the existing knowledge base were used to analyze these events. During the analysis of the new loops, we performed some modifications on 14 plans. The modifications were limited to minor adjustments that increase the accuracy of the plans. Examples of these modifications are: changing the descriptive name of a plan, modifying a plan firing condition, and writing parts of a plan antecedent or consequent in more general forms. Performing such modifications is a normal outcome of the experience gained from the additional and monitored usage of the knowledge base. No modifications were performed on the tree structure of a plan to analyze additional programming concepts. Moreover, no plan was rewritten to analyze a different concept. The modified plans analyze the same concepts as before but in a more accurate manner.

3.3 Results and analysis

Table 2 shows the number of loops in all the programs. Each of the new programs is smaller than the original one with respect to the number of loops it contains and its executable source lines of code (SLOC). Choosing this collection of small and different programs ensures that the newly analyzed loops are of varying characteristics and that they are different in their designs and functions from the original set of loops.

Table 3 compares between the completely analyzed, partially analyzed, and unanalyzed loops with respect to

Table 2. Number of loops in the programs analyzed.

Program	SLOC (Executable)	number of loops
Original program (scheduling)	1400	77
Set of general purpose loops	—	30
Scanner I	483	22
Scanner II	654	19
Student records	236	12
Linked list system	277	9
Total (over newly analyzed loops)		92

two measures of the loop size. These measures are the average number of executable source lines of code (SLOC) and the average number of events per loop. In the original study, the sizes of the partially analyzed and unanalyzed loops were considerably higher than that of the analyzed ones. However, this large variation in size is not consistent across the newly analyzed loops. The reason for this is that the original program was analyzed to create the knowledge base. While plans were designed for the analysis of most loops, difficulties were encountered in formally analyzing large and complicated loops and designing their plans. Because the new loops were analyzed using the existing set of plans, some events were unanalyzed because of the unavailability of appropriate plans in the knowledge base. This unavailability is not necessarily due to the difficulty of performing the formal analysis needed to design the plans.

Tables 4-6 show the results needed to further test the first two analysis hypotheses given at the beginning of this section. Given that the knowledge base only contained 48 plans designed during the analysis of the original program, the percentages of analyzed loops and events generally show a good usability of the knowledge base beyond the original program.

Results in Table 4 support the hypothesis that loop complexity dimensions are valid indicators of its amenability to analysis. That is, the loop classes in the middle column of Table 1 are more amenable to analysis than those in the right column. It should be noted, however, that the reasons for the difficulty of analyzing the classes in the right column are different. In case of

the original program, it was more difficult to formally analyze the loop classes in the right column and design their plans. In the new program, the difficulty stems from the fact that plans for analyzing classes in the right column were less common in the existing knowledge base. As we pointed out earlier, this scarcity is not necessarily due to the difficulty of designing the plans.

The percentages of analyzed loops and events shown in Tables 5 and 6 generally support the second hypothesis given at the beginning of this section. That is, the loop decomposition and plan design methods of our approach can make the plans applicable in many different loops and across different applications. The highest percentage of analyzed loops and events occur in the set of general purpose loops. The explanation for this is that a user manual usually contains many stereotyped programming concepts which were captured by the plans in our knowledge base. The other programs contain more application specific loops which could not be analyzed by the existing set of plans. The lowest percentage of analyzed loops and events occur in the linked list system and scanner II. Although the plans in the knowledge base handled pointers, they did not cover specific concepts like deleting nodes and sorting a linked list. This explains the low percentage of analyzed events in the linked list system. As for scanner II, it contained several loops for building and accessing symbol tables and reading and building deterministic finite state automata. These loops are specific to the application domain of scanners and plans which analyze them were certainly unavailable in the existing knowledge base.

Table 3. Comparison between the completely analyzed, partially analyzed, and unanalyzed loops.

Program	Average number of executable SLOC / loop (SD)			Average number of events / loop (SD)		
	completely analyzed	partially analyzed	unanalyzed	completely analyzed	partially analyzed	unanalyzed
Original Program (scheduling)	10.5 (8.3)	48.8 (22.9)	40.4 (9.0)	3.3 (2.1)	14.0 (6.0)	10.9 (3.6)
Set of general purpose loops	7.4 (5.1)	16.8 (5.6)	14.8 (12.0)	3.0 (2.2)	5.3 (1.8)	2.8 (0.4)
Scanner I	5.3 (3.8)	12.2 (7.2)	9.8 (8.1)	1.9 (0.7)	4.6 (2.0)	6.8 (5.7)
Scanner II	11.2 (8.3)	7.5 (1.5)	16.8 (7.9)	2.3 (0.6)	3.0 (1.0)	12.5 (6.0)
Student Records	11.1 (5.6)	23.0 (17.0)	6.0 (1.4)	2.3 (0.5)	7.5 (4.5)	4.0 (1.4)
Linked List system	5.0 (1.6)	14.0 (0.0)	14.0 (5.2)	1.8 (0.4)	4.0 (0.0)	5.3 (1.1)

Table 4. Number of loops analyzed along the three classification dimensions.

Analysis statistics	Dimension					
	1		2		3	
	Simple loop	General loop	Noncomposite condition	Composite condition	Flat body	Nested body
Number of loops in the original program	52	25	46	31	53	24
Number of completely analyzed loops	48 (92%)	17 (68%)	42 (91%)	23 (74%)	52 (98%)	13 (54%)
Number of loops in the new programs	57	35	69	23	76	16
Number of completely analyzed loops	46 (81%)	11 (31%)	45 (65%)	12 (52%)	51 (67%)	6 (38%)

Table 5. Number of analyzed and unanalyzed events.

Program	number of events	analyzed events	unanalyzed events
Original Program (scheduling)	356	235 (66%)	121 (34%)
Set of general purpose loops	98	73 (74.5%)	25 (25.5%)
Scanner I	75	41 (54.7%)	34 (45.3%)
Scanner II	106	27 (25.5%)	79 (74.5%)
Student Records	43	18 (41.9%)	25 (58.1%)
Linked List system	32	9 (28%)	23 (72%)
Total (over newly analyzed loops)	354	168 (47.5%)	186 (52.5%)

Table 6. Number of completely analyzed, partially analyzed, and unanalyzed loops.

Program	number of loops	completely analyzed loops	partially analyzed loops	unanalyzed loops
Original Program (scheduling)	77	65 (84.4%)	4 (5.2%)	8 (10.4%)
Set of general purpose loops	30	22 (73.3%)	4 (13.3%)	4 (13.3%)
Scanner I	22	13 (59.1%)	5 (22.7%)	4 (18.2%)
Scanner II	19	11 (57.9%)	2 (10.5%)	6 (31.6%)
Student Records	12	7 (58.3%)	2 (16.7%)	3 (25%)
Linked List system	9	4 (44.4%)	1 (11.1%)	4 (44.4%)
Total (over newly analyzed loops)	92	57 (62%)	14 (15.2%)	21 (22.8%)

Table 7. Original and additional utilization of the original set of plans.

Name (subscript)	Plan category											
	DBP		IBP		SLAP		GLAP		SHAP		GHAP	
	O*	A†	O*	A†	O*	A†	O*	A†	O*	A†	O*	A†
1	45	37	4	5	23	32	4	1	3	4	3	2
2	6	9	15	13	19	13	13	2	13	7	2	0
3	8	8	1	3	3	0	1	0	1	1	—	—
4	9	4	2	0	1	5	2	0	1	1	—	—
5	1	1	2	0	1	1	1	0	1	0	—	—
6	—	—	2	0	1	1	1	0	1	0	—	—
7	—	—	—	—	1	1	1	0	2	0	—	—
8	—	—	—	—	20	15	1	0	2	0	—	—
9	—	—	—	—	3	0	—	—	2	0	—	—
10	—	—	—	—	—	—	—	—	1	0	—	—
11	—	—	—	—	—	—	—	—	2	0	—	—
12	—	—	—	—	—	—	—	—	1	0	—	—
13	—	—	—	—	—	—	—	—	1	0	—	—
14	—	—	—	—	—	—	—	—	1	0	—	—
15	—	—	—	—	—	—	—	—	2	0	—	—
16	—	—	—	—	—	—	—	—	1	0	—	—
17	—	—	—	—	—	—	—	—	3	1	—	—
18	—	—	—	—	—	—	—	—	1	1	—	—
Total	69	59	26	21	72	68	24	3	39	15	5	2

O* Original A† Additional

Table 7 shows the results needed to test the new analysis hypothesis given at the beginning of this section. It shows the utilization of the original set of 48 plans during the analysis of the original program. It also shows the additional utilization of the same plans in analyzing the new programs. 24 out of the 48 plans were used in analyzing the new loops. We notice from these results that the plans that were highly used originally were also highly used in the new programs (e.g., DBP₁, SLAP₁, SLAP₂, SHAP₂). Most of the plans that were used once in the original program never got used in the new programs. That is, many of the plans that were highly used in the original study represent stereotyped programming concepts that are commonly used across different applications.

4 Conclusions

In this paper, we performed an evaluation of a knowledge-based approach to the analysis of loops. This evaluation served to validate several hypotheses related

to the analysis approach. By using plans in an existing knowledge base to analyze five different programs, we demonstrated that the knowledge base generated for a given program is generally usable beyond that original program.

Future research includes experimenting with the analysis approach on programs that are larger than the ones considered so far. The practicality of our approach can be greatly enhanced by trying to create knowledge bases that are sufficient for specific application domains. This can also serve to address and investigate several issues related to the acquisition and development of plans and the generality and efficiency of our analysis approach with respect to different application domains.

References

- [1] Abd-El-Hafiz, S. K. and Basili, V. R. A knowledge-based approach to the analysis of loops. *IEEE Trans. on Software Engineering*, 22(5):339-360, 1996.

- [2] Abd-El-Hafiz, S. K. and Basili, V. R. *A knowledge-based approach to program understanding*. Kluwer Academic Publishers, 1995.
- [3] Abd-El-Hafiz, S. K. and Basili, V. R. A tool for assisting the understanding and formal development of software. *Proceedings of the Sixth International Conference on Software Engineering and Knowledge Engineering*, Jurmala, Latvia, pages 36-45, 1994.
- [4] Abd-El-Hafiz, S. K. and Basili, V. R. Documenting programs using a library of tree structured plans. *Proceedings of the Conference on Software Maintenance*, Montreal, Canada, pages 152-161, 1993.
- [5] Abd-El-Hafiz, S. K., Basili, V. R., and Caldiera, G. Towards automated support for extraction of reusable components. *Proceedings of the Conference on Software Maintenance*, Sorrento, Italy, pages 212-219, 1991.
- [6] Barth, C. W. Table-driven scanner, the compiler project for CMSC 430 course offered at the University of Maryland, College Park, MD, 1985.
- [7] Basili, V. R. and Abd-El-Hafiz, S. K. A method for documenting code components. *The Journal of Systems and Software* (to appear).
- [8] Gries, D. *The science of programming*. Springer-Verlag, 1981.
- [9] Harandi, M. T. and Ning, J. Q. Knowledge-based program analysis. *IEEE Software*, 7(1):74-81, 1990.
- [10] Hartman, J. Understanding natural programs using proper decomposition. *Proceedings of the 13th International Conference on Software Engineering*, pages 62-73, Austin, Texas, 1991.
- [11] Hoare, C. A. R. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576-580, 583, 1969.
- [12] Jalote, P. *An integrated approach to software engineering*. Springer-Verlag, 1991.
- [13] Johnson, W. L. and Soloway, E. PROUST: knowledge-based program understanding. *IEEE Trans. on Software Engineering*, SE-11(3):267-275, 1985.
- [14] Lamie, E. L. *Pascal Programming*. John Wiley, 1987.
- [15] Letovsky, S. Program understanding with the lambda calculus. *Proceedings of the 10th International Joint Conference on AI*, pages 512-514, 1987.
- [16] Mills, H. D., Basili, V. R., Gannon, J. D., and Hamlet, R. G. *Principles of computer programming: a mathematical approach*. Allyn and Bacon, 1987.
- [17] Norvig, P. *Paradigms of artificial intelligence programming: case studies in Common Lisp*. Morgan Kaufmann, 1992.
- [18] Quilici, A. A hybrid approach to recognizing programming plans. *Proceedings of the Working Conference on Reverse Engineering*, pages 126-133, Baltimore, Maryland, 1993.
- [19] Rich, C. and Wills, L. M. Recognizing a program's design: a graph-parsing approach. *IEEE Software*, 7(1):82-89, 1990.
- [20] Rich, E. and Knight, K. *Artificial intelligence*. McGraw-Hill, 1991.
- [21] Schneider and Bruell. *Advanced programming and problem solving with Pascal*. John Wiley, 1987.
- [22] Waters, R. C. A method for analyzing loop programs. *IEEE Trans. on Software Engineering*, SE-5(3):237-247, 1979.
- [23] Jensen, K. and Wirth, N. *Pascal user manual and report*. Springer-Verlag, 1985.