A Knowledge-Based Approach to the Analysis of Loops

Salwa K. Abd-El-Hafiz, *Member, IEEE Computer Society*, and Victor R. Basili, *Fellow, IEEE*

Abstract—This paper presents a knowledge-based analysis approach that generates first order predicate logic annotations of loops. A classification of loops according to their complexity levels is presented. Based on this taxonomy, variations on the basic analysis approach that best fit each of the different classes are described. In general, mechanical annotation of loops is performed by first decomposing them using data flow analysis. This decomposition encapsulates closely related statements in events, that can be analyzed individually. Specifications of the resulting loop events are then obtained by utilizing patterns, called plans, stored in a knowledge base. Finally, a consistent and rigorous functional abstraction of the whole loop is synthesized from the specifications of its individual events. To test the analysis techniques and to assess their effectiveness, a case study was performed on an existing program of reasonable size. Results concerning the analyzed loops and the plans designed for them are given.

Index Terms—First order predicate logic, formal specifications, knowledge base, loops, program understanding, reverse engineering.

1 Introduction

Program understanding plays an important role in nearly all software related tasks. It is vital to the maintenance and reuse activities. Such activities cannot be performed without a deep and correct understanding of the component to be maintained or reused. Program understanding is also indispensable for improving the quality of software development activities such as code reviews, debugging, and some testing approaches. All these development activities require programmers to read and understand programs.

Due to the importance of program understanding, there has been considerable research on techniques and tools for analyzing and understanding computer programs. Within these efforts, substantial interest is usually directed towards the specific topic of analyzing loops. This interest stems mainly from inherent reasoning difficulties involving repeated program state modifications and the fact that loops have a major effect on program understandability [42].

To analyze loops and reason about their properties, some approaches define heuristics that can be used to guide a search for a loop invariant [19] or function [32]. However, heuristic techniques in general are not always useful. After applying the heuristics a considerable number of times, one may or may not succeed in finding a correct invariant or function. Other approaches focus on developing algorithmic techniques for finding the invariants or functions of specific simple classes of loops. The research performed by Basu and Misra [8], Dunlop and Basili [12], Katz and Manna [26], and Wegbreit [49] is representative of these loop analysis approaches. These algorithmic approaches analyze loops through the use of formal, semantically sound, and unambiguous notation. Although some of them provide guidelines on how to mechanically generate loop invariants or functions, no algorithmic techniques were actually used to implement automatic analysis systems. A different approach, that analyzes loops by mechanically decomposing them into smaller fragments, was adopted by Waters [47]. Even though Waters' approach does not address the issue of how to use this decomposition to mechanically annotate loops, it is especially interesting because of its practicality.

To analyze complete programs, the knowledge-based approaches utilize a knowledge base of plans in providing intelligent analysis results. Plans are defined as units of knowledge representing, or necessary for identifying, abstract programming concepts [15], [16], [24], [37], [38], [48]. These approaches are inspired by the cognitive studies [31], [41], [43] which suggest that the understanding process is one in which programmers make use of stereotyped solutions to problems in making sophisticated high-level decisions about a program. These knowledge-based approaches are all implemented, to varying degrees, in automatic analysis systems. Some of these approaches are: graph-parsing [38], [50]; top-down analysis using the program's goals as input [23], [24]; top-down analysis using a functional representation of programs that relates the program code and goals to a proof of correctness [6], [33]; heuristic-based object-oriented recognition [15], [16]; transformation of a program into a semantically equivalent but more abstract form with the help of plans and transformation rules [27], [29], [46]; and decomposition of a program into smaller more tractable parts using control flow analysis [17] or program slicing [18]. Even though these approaches demonstrate the feasibility and usefulness of the automation of program understanding, they lack some important features.

Most of the knowledge-based program analysis and understanding approaches produce program documentation

- *S.K. Abd-El-Hafiz is with the Engineering Mathematics Department, Faculty of Engineering, Cairo University, Giza, Egypt. E-mail: elhafiz@cairo.eun.edu.*
- *V.R. Basili is with the Department of Computer Science, University of Maryland, College Park, MD 20742. E-mail: basili@cs.umd.edu.*

Manuscript received Mar. 11, 1994; revised Jan. 19, 1996.
Recommended for acceptance by D. Wile.
For information on obtaining reprints of this article, please send e-mail to: transse@computer.org, and reference IEEECS Log Number S95491.

that is generally in the form of structured natural language text [9], [15], [16], [17], [24], [36], [38], [50]. Such informal documentation gives expressive and intuitive descriptions of the code. However, there is no semantic basis that makes it possible to determine whether or not the documentation has the desired meaning. This lack of a firm semantic basis makes informal natural language documentation inherently ambiguous.

Some of the knowledge-based approaches rely on real-time user-supplied information that might not be available at all times. For instance, goals a program is supposed to achieve [6], [24] or transformation rules that are appropriate for analyzing a specific code fragment [27], [46] are not always clear to the user. Others have difficulty in analyzing nonadjacent program statements [29]. In addition, a significant amount of program analysis and understanding research used toy programs that are less than 100 lines of code to validate proposed approaches. Realistic evaluations of these approaches, which give quantifiable results about recognizable and unrecognizable concepts in real and existing programs, are needed. Such evaluations can also serve as a basis for empirical studies and future comparisons with other approaches [40].

To address the above-mentioned drawbacks, we present a knowledge-based approach to the automation of program analysis. It combines and builds on the strengths of a practical program decomposition method [47], the axiomatic correctness notation [19], and the knowledge-based analysis approaches. It mechanically documents programs by generating first order predicate logic annotations of their loops. The advantages of predicate logic annotations are that they are unambiguous and have a sound mathematical basis. This allows correctness conditions to be stated and verified, if desired. Another advantage is that they can be used in assisting formal development of software using such languages as VDM and Z [25], [45].

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. The first dimension focuses on the control computation of the loop. As defined by Pratt [35], the control computation for a loop is that part concerned with the initialization, modification, and testing of the variables which determine the flow of control into, through, and out of the loop. The second dimension focuses on the complexity of the loop condition as determined by the number of clauses it has. The third dimension focuses on the complexity of the loop body. Based on this taxonomy, the analysis techniques that can be applied to the different loop classes are described.

In general, we annotate loops with predicate logic assertions in a step-by-step process as depicted in Fig. 1 [1]. The analysis of a loop starts by decomposing it into fragments, called *events*. Each event encapsulates the loop parts that are closely related, 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.

This study tests several hypotheses related to the presented analysis approach:

- A loop complexity dimensions are indicators of its amenability to analysis.
- The loop decomposition and plan design methods can make the plans applicable to many loops that are different in their designs and functions. This, in turn, can increase plan utilization.
- The analysis techniques can be automated.

To test the first two hypotheses and to characterize the practical limits of the analysis approach, a case study on a set of 77 loops in an existing Pascal program for scheduling university courses has been performed. The program has 1,400 executable lines of code and the loops analyzed have the usual programming language features such as pointers, procedure and function calls, and nested loops. However, the loops analyzed do not involve recursive function and procedure calls. Recursion is not currently being handled by our analysis approach. To test the third hypothesis, a prototype tool, which annotates loops with predicate logic annotations, was developed.

Section 2 of this paper gives some of the definitions used. Section 3 introduces the loop taxonomy. Sections 4 and 5 describe the techniques used for analyzing flat and nested loops, respectively. Section 6 discusses the approach presented and highlights its advantages and limitations. Section 7 describes how the case study was performed and gives the results of the analysis. Section 8 briefly explains the design and structure of the implemented prototype tool. Finally, conclusions and future research directions are given in Section 9. Appendices A and B give the notation and acronyms used throughout the rest of the paper.

2 Definitions

We start by defining some of the notation used throughout this paper. First, we give the definitions related to the representation of while loops.

A *control-flow graph* is a directed graph that has one node for each simple statement and one node for each control predicate. There is an edge from node I to node J if an execution of J can immediately follow that for I [21].

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. The sequence of values *scanned* by a control variable are these values that get assigned to the control variable and actually used in the loop body.

Now, we give some definitions that introduce the language and terminology used in the analysis. A *concurrent assignment* is a statement in which several variables can be assigned simultaneously. We use the form $v_1, v_2, \ldots, v_n := e_1,$

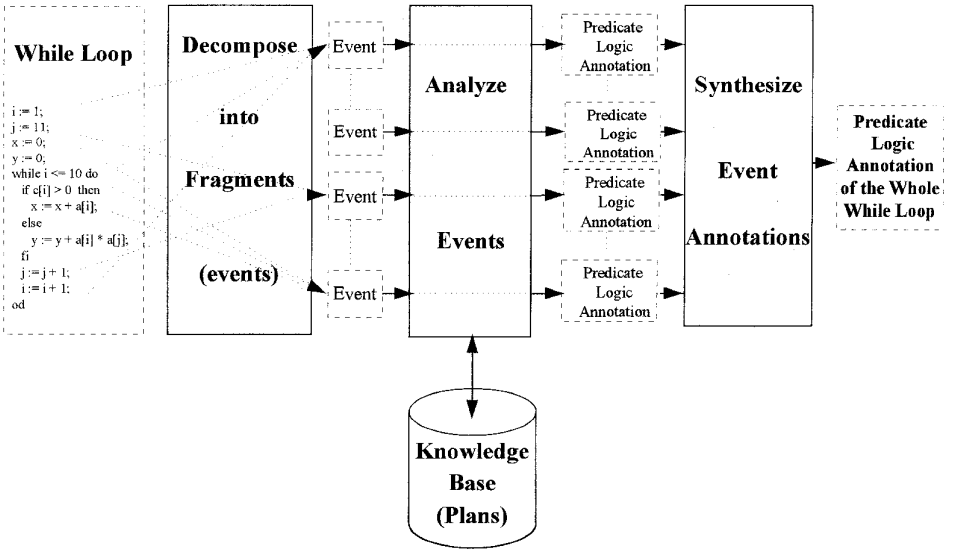

Fig. 1. Overview of the analysis approach.

$e_2, \ldots, e_n$ to assign every ith expression from the right hand list to its corresponding ith variable from the left hand list [14], [32]. 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 [14], [32]. Similar to Gries' definition of the alternative command, all the guards must be well defined [14]. However, it is possible that all guards evaluate to false. In this case, no variable is modified (i.e., the conditional assignment evaluates to a skip command [14]). It should also be noted that because we are only analyzing deterministic programs, all the guards are mutually exclusive.

Any variable assigned in a conditional assignment defines the *data flow out* of the statement. Any variable referenced by a conditional assignment defines the *data flow into* the statement. Two conditional assignments are said to be *circularly dependent* if some variable is responsible for data flow out of one statement and into the other, either directly or indirectly, and vice versa.

3 A 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, the 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 can vary between two cases. In the *noncomposite* case, B is a logical expression that consists of one clause of the conjunctive normal form [39]. 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 cannot include other loops. In nested structures, however, the loop body includes one or more loops.

TABLE 1
THE THREE DIMENSIONS USED FOR CLASSIFYING LOOPS

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

4 Analysis of Flat Loops

As depicted in Fig. 2, the analysis of flat loops is performed in a step-by-step process divided into four main phases. Descriptions of these phases and their application to the example shown in Fig. 3 are given in the remainder of this section [3]. In this example, a simple loop with a noncomposite condition scans a segment of the array *capacity* searching for its minimum.

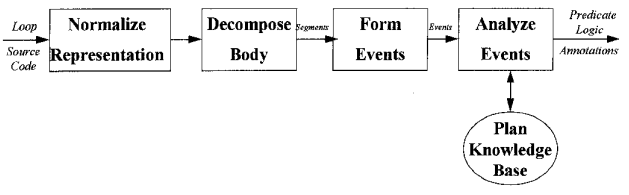

Fig. 2. Analysis of flat loops.

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

Fig. 3. Analysis of flat loops.

4.1 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.

Normalization of the Loop Condition. The loop condition is converted into a standard normal form, which is the *conjunctive normal form*. This normal form represents a well-formed formula (wff) in predicate logic as a conjunction of clauses where a clause is defined to be a wff in conjunctive normal form but with no instances of the *and* connector [39]. For example, the loop condition $x < a$ *or* ($y < b$ and $z < c$) is transformed to the conjunction of two clauses. The first clause is ($x < a$ *or* $y < b$) and the second is ($x < a$ *or* $z < c$).

Normalization of the Loop Body. A single unwinding of the loop body is performed by symbolic execution [4] 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 simplifications. Arithmetic expressions are simplified by converting them into an internal canonical form for polynomials, manipulating them, and converting them back to their external form [34]. Predicate simplifications are performed using rule-based transformations. Since the simplification details are dependent on our specific prototype implementation, they are not discussed during the description of the analysis phases.

For the loop given in Fig. 3, the condition is already in conjunctive normal form containing the one clause $j <$ *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:

Name	Conditional Assignment
C_1	$capacity[j] < min \Rightarrow index := j,$
C_2	$capacity[j] < min \Rightarrow min := capacity[j],$
C_3	$true \Rightarrow j := j + 1$

4.2 Decomposition of the Loop Body

To facilitate the mechanical generation of loop annotations, the symbolic execution result is uniquely decomposed into *segments* of code that can be analyzed separately. Each segment encapsulates the statements that are interdependent with respect to data flow. The *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.

To obtain the loop segments, we assume that the conditional assignments of the symbolic execution result correspond to the nodes of a directed graph. An edge from node C_i to node C_k exists if and only if there is data flowing out of C_i into C_k and C_i and C_k are distinct. The strongly connected components of this graph correspond to the loop segments [10].

For the loop shown in Fig. 3, the three conditional assignments of the symbolic execution result form a directed graph G with three edges: two from C_3 to C_1 and C_2, and one from C_2 to C_1. Since there are no cycles in G, its strongly connected components correspond to its nodes. Thus, the loop segments correspond to C_1, C_2, and C_3.

Because the analysis of a segment might be dependent on the analysis results of other segments, a segment analysis result should be obtained before analyzing the segments dependent on it. That is why we need to order the segments according to their data flow dependencies [47]. Assuming that S is the set of segments in the loop body, the order of each segment is determined by the following algorithm:

1) Set m to 1.
2) While the number of segments in S is ≥ 1 do
   a) Identify the maximal subset of S such that each of its segments does not have data flowing out into other segments of S.
   b) Let the order of the identified segments be m.
   c) Remove the identified segments from S.
   d) Increment m.
3) Let the final order of each segment be (m – old order).

Step 2 of the above algorithm assigns unique orders to the segments such that order of $S_i >$ order of S_k if and only if there is data flowing, either directly or indirectly, from segment S_i to segment S_k. Step 3 produces an irreflexive partial order of the segments. The resulting ordering relation 'analyzed before,' is denoted by '$\rightarrow$'. It is irreflexive because it is meaningless for a segment to be analyzed before itself. It satisfies the antisymmetric property because any two distinct segments, by definition, have no circular dependencies. The design of the above algorithm ensures the satisfaction of the transitive property. Moreover, it is possible for two segments to be unrelated (i.e., they can have the same order).

In the example given in Fig. 3, let the segments of the loop be S_1, S_2, and S_3 that correspond to C_1, C_2, and C_3, respectively. The orders assigned to these segments using the above algorithm are:

Order	Name	Segment
1	S_3	$j := j + 1$
2	S_2	$capacity[j] < min \Rightarrow min := capacity[j]$
3	S_1	$capacity[j] < min \Rightarrow index := j$

Notice that the segment that defines j, S_3, has the lowest order because the other two segments, S_1 and S_2, reference j (i.e., $S_3 \rightarrow S_1$ and $S_3 \rightarrow S_2$). Similarly, $S_2 \rightarrow S_1$ because *min* is defined in S_2 and referenced in S_1. Since the premise of the conditional assignment that modifies j is *true*, it is removed.

4.3 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*.

To form BEs, each clause of the loop condition is used as the condition of a unique BE. Then, the enumeration of each BE is constructed from the highest order segment(s) having data flow into the condition. If a clause has no segment responsible for the data flow into it, this means that this clause is redundant and should be removed from the loop condition. If a segment is responsible for the data flow into the loop condition but remains with no clause associated with it, this segment is used as the enumeration of a new BE whose condition is set to *true*. The initializations of the control variables defined in a BE are included in the initialization part.

The BE of the loop given in Fig. 3 is formed by combining the unique condition clause, (j < *num_of_rooms* + 1), with the only segment that is responsible for the data flow into it, S_3. Since the loop under consideration has no initializations, we use the notation *j?* to denote the initial value of the variable j. As a result, the BE has the following form:

condition: $j <$ *num_of_rooms* + 1
enumeration: $j := j + 1$
initialization: $j :=$ *j?*

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*.

After identifying the BEs, the AEs bodies are formed from the segments of the loop that did not get used in BEs. The initialization of each variable defined in an AE is then included in it.

For the loop shown in Fig. 3, the remaining segments S_2 and S_1 constitute the bodies of two AEs given below. The notation *min?* and *index?* are used to denote the initial values of the variables *min* and *index*

1) AE 1
   body: *capacity*[j] < *min* ⇒ *min* := *capacity*[j]
   initialization: *min* := *min?*
2) AE 2
   body: *capacity*[j] < *min* ⇒ *index* := j
   initialization: *index* := *index?*

Finally, 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 events of the loop shown in Fig. 3 are thus ordered as follows.

1) BE (order 1)
   condition: $j <$ *num_of_rooms* + 1
   enumeration: $j := j + 1$
   initialization: $j :=$ *j?*
2) AE (order 2)
   body: *capacity*[j] < *min* ⇒ *min* := *capacity*[j]
   initialization: *min* := *min?*
3) AE (order 3)
   body: *capacity*[j]< *min* ⇒ *index* := j
   initialization: *index* := *index?*

4.4 A Knowledge Base of Plans

To analyze the loop events, we utilize plans stored in a knowledge base. We use the term 'plan' to refer to a unit of knowledge required to identify an abstract concept in a program. Our plans are used as inference rules [15], [16]. Their basic structure is divided into two parts: the antecedent and the consequent. When a loop event matches a plan antecedent, the plan is fired. The instantiation of the information in the consequent represents the contribution of this plan to the loop specifications. To guarantee the accuracy of the predicate logic specifications included in the consequents, no partial matches with antecedents are allowed (i.e., the antecedent has to be completely matched).

The knowledge base is designed so that any two plans do not have similar antecedents. Thus, a loop event can only match the antecedent of a unique plan. It should also be noted that the possibility of designing as many plans as the number of loop events in a specific program is reduced because the loop events encapsulate abstract concepts that can occur in different loops. Section 7 will examine this issue of the knowledge base size in more detail.

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.

In case of simple loops, the sequences of values scanned by the control variable during and after the execution of a simple loop can be easily written because the control computation is isolated from the rest of the loop. The loop condition, the control variable's initial value, and the net modification performed on the control variable in one loop iteration, if any, provide sufficient information for writing these sequences. This specific information about the control computation of the loop can be used to produce equally specific loop specifications. The plans that analyze simple loops can include these sequences and utilize them in writing the loop specifications. The loop specifications produced for simple loops are the preconditions, invariants, and postconditions. The formal approach used for deriving the invariants is the axiomatic approach [14], [19], [20]. In this approach, if we assume that B, S, S_0, I, P, and Q are the loop condition, body, initialization, invariant, precondition, and postcondition, respectively, then the relations between

them are given in the following rules. In these rules, the notation $P\{S\}Q$ means that if the predicate P is true before executing the first statement of the program part S, and if S terminates, then the predicate Q will be true after execution of S is complete.

I {while B do S} I *and* $\neg B$,
(I *and* $\neg B \Rightarrow Q$), and
($P \Rightarrow T$), where T is deduced from $T\{S_0\}I$.

The analysis of *general loops* is not as straightforward as that of simple ones. In many cases, it might not be easy, or even possible, to obtain such specific knowledge because the control computation of the loop is not as determinate and isolated as in the case of simple loops. The sequences of values scanned by the control variable(s) and the program state at the end of the loop are usually dependent on the combined indeterminate effects of several events and the values of some program variables. As a result, the plans that analyze general loops neither include the aforementioned sequences nor utilize them in writing the loop specifications. The loop postcondition can only be deduced after the synthesis of the loop invariant. The postcondition is formed by taking the conjunction of the loop invariant with the negation of the loop condition [14], [19]. Using this method to obtain the loop postcondition yields predicates that might not be as informative and concise as those of simple loops. As a result, additional simplifications might be needed to reduce the complexity and improve the readability of general loops postconditions.

For instance, consider the simple loop shown in Fig. 3. The sequence scanned by the control variable at any point during the loop execution is *j?* to $j - 1$. This sequence is needed to write the part of the invariant:

$$min = MIN(\{min?\} \cup \{capacity[j? \,..\, j-1]\}),$$

where *MIN*(s) is the minimum of the set s and $\cup$ is the set union operator. The final sequence of values scanned by the control variable in this loop is *j?* to *num_of_rooms*. This sequence is needed to write the part of the postcondition:

$$min = MIN(\{min?\} \cup \{capacity[j? \,..\, num\_of\_rooms\,]\}).$$

In the general loop given in Fig. 4, however, there is no guarantee that the final sequence scanned by the control variable j will be *j?* to *num_of_rooms*. The value of the final sequence is dependent on the interaction of the two events that modify *flag* and j, and the contents of the variables *capacity* and *limit*. As a result of this generality of the control computation, the sequences of values scanned by the control variable(s) and, consequently, the postcondition parts of the individual events cannot be written.

```
while (j <= num_of_rooms + 1) and (flag = false) do begin
    if capacity[j] < limit then begin
        index := j;
        flag := true
    end;
    j := j + 1
end
```

Fig. 4. Example of a general loop.

To accommodate the differences between simple and general loops, we have two categories of BPs. Determinate BPs (DBPs) contain in their consequents information regarding the postcondition and the sequences of values scanned by the control variable. Indeterminate BPs (IBPs), on the other hand, do not contain such information. We also have two categories of APs. Simple APs (SAPs) utilize the above sequences in writing the loop specifications, including its postcondition. General APs (GAPs) do not include the loop postcondition part or utilize the above sequences. These plan categories are shown in Fig. 5. It should be noticed that because the information contained in the consequents of IBPs is a subset of that contained in the consequents of DBPs, DBPs can be used in analyzing general loops. In such cases, we neglect the information regarding the control sequences and the postcondition in the DBPs consequents. However, because IBPs consequents do not contain such specific information, IBPs cannot be used in analyzing simple loops.

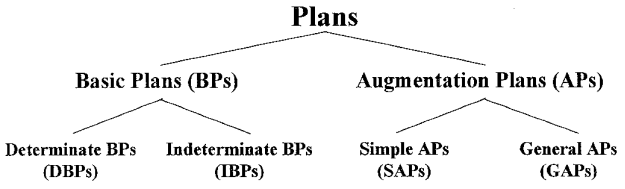

Fig. 5. Plan categories.

In general, The information included in a plans antecedent and consequent are described below. In this description, the words printed in **bold** correspond to fields in the plans (see Figs. 6 and 7).

An antecedent contains the following information:

1) An individual listing of the control variables, in the **control-variables** part, which serves to underscore their importance and to facilitate the design, readability, and comprehension of the plan.
2) Generic patterns of BEs and AEs that are used to match stereotyped loop events.
3) Knowledge needed for the correct identification of the plans such as data taype informaiton and the results of analyzing previous events. This knowledge is given in the **firing-condition**.

A consequent includes the following information:

1) Knowledge necessary for the annotation of loops with their Hoare-style [19] specifications. The **precondition** and **invariant** have the usual meaning [19]. The **postcondition** part gives information, in case of simple loops, about the variables values after the loop execution ends. It is correct provided that the loop executes at least once. If the loop does not execute, no variable gets modified.
2) In case of DBPs, knowledge about the sequence of values scanned by the control variables at any point during and after the loop execution is captured in **sequence** and **final-sequence**, respectively.

Fig. 6 and Fig. 7 show two example plans of the categories DBP and SAP, respectively. To convey the basic analysis

ideas within a reasonable space limit, we only show simplified versions of the plans. The suffix '#' is used to indicate terms in the antecedent (or consequent) that must be matched (or instantiated) with actual values in the loop events.

plan-name	DBP_1 (ascending enumeration)
antecedent	
control-variables	*var#*
condition	*var# R# exp#*
enumeration	*var# := SUCC(var#)*
initialization	*var# := var?#*
firing-condition	(*R#* is relational operator that equals ≤ or <) *and* (*var#* is of a discrete ordinal type) *and* (Noncomposite or general loop condition)
consequent	
precondition	*PRED(var?#) R# exp#*
invariant	*var?# ≤ var# R# SUCC(exp#)*
postcondition	*var# = SUCC(SHIFT(exp#))*
sequence	*var?# .. PRED(var#)*
final-sequence	*var?# .. SHIFT(exp#)*
inner-addition	*var?# ≤ var# R# exp#*
where,	
i .. j	Sequence of integers from *i* up to *j inclusive.*
SUCC (x)	The successor of *x*.
PRED (x)	The predecessor of *x*.
SHIFT	The identity function if *R#* equals ≤. Equals *PRED* otherwise.

Fig. 6. A determinate basic plan.

plan-name	SAP_5 (find minimum)
antecedent	
control-variables	*v#*
body	*a#[exp#] R# lhs# ⇒ lhs# := a#[exp#]*
initialization	*lhs := lhs?#*
firing-condition	(*R#* equals ≤ or <)
consequent	
precondition	*true*
invariant	$lhs = MIN(\{lhs?\#\} \cup \{a\#[exp\#\ ^{v\#}_{\textbf{sequence}}\ ]\})$
postcondition	$lhs = MIN(\{lhs?\#\} \cup \{a\#[exp\#\ ^{v\#}_{\textbf{final-sequence}}\ ]\})$
inner-addition	Same as invariant.
where,	
MIN(s)	The minimum of the set *s*.

Fig. 7. A simple augmentation plan.

The plan DBP_1 (Fig. 6) represents an enumeration construct that goes over a sequence of values of a discrete ordinal type in an ascending order with a unit step. In the case where the loop has a composite condition, the **sequence**, **final-sequence** and **postcondition** of this plan are written in a more general form that enables deducing the corresponding **sequence**, **final-sequence** and **postcondition** of the loop from the multiple BEs it contains. The plan SAP_5 (Fig. 7) searches for the minimum of a segment of the array *a#* and stores it in the variable *lhs#*.

The knowledge base in a specific application domain should be created by an expert in both formal specifications and this domain. The expert should analyze the commonly used events in this domain and create new plans or improve on already existing ones. In creating this knowledge base, its size should be controlled by increasing the utilization of the designed plans. The loop decomposition method was designed for this purpose; to reveal the common algorithmic constructs that can be incorporated in many different loops. The hypothesis is that this decomposition can have a positive effect on plan utilization and, hence, on the size of the knowledge base. Improvements on the structure and/or the knowledge represented in the plans can also make the plans applicable to a larger set of events.

Knowledge representation improvements, called *abstractions*, involve replacing some of the terms in a plan with more abstract ones that make the plan capable of analyzing more cases. For example, replacing the addition operator, +, in a plan that analyzes an accumulation by summation event by a more abstract one that denotes either addition or multiplication represents an abstraction of this plan. The new plan can analyze both accumulation by summation and accumulation by multiplication events.

Structural improvements to a plan modify the basic structure into a tree structure that allows the inclusion of several similar plans in one tree-structured plan. 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 other words, a tree-structured plan consists of a single antecedent and several consequents organized into one or more tree structures as shown in Fig. 8. The consequents are organized into one tree if the default consequent exists. Otherwise, they are organized into more than one tree (forest). 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 Fig. 8 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. When the event satisfies the antecedent, the search for the appropriate consequent starts at the appropriate root going down in the tree as far as possible. The edge between a parent and a child can only be taken if the local **firing-condition** associated with this edge is satisfied.

Tree-structured plans can be used to detect special cases and output loop specifications that are simple and concise. They can also be used to analyze similar events whose specifications vary depending on their environment (e.g., data types, control computation of the loop, ..., etc.).

For instance, the plan SAP_5 (Fig. 7) can be structurally improved as shown in Fig. 9. The antecedent is similar to that shown in Fig. 7 except for the firing condition. The antecedent **firing-condition** now allows *R#* to be matched with more relational operators. Three local **firing-conditions** and the consequents cover three different variations. Consequent 1, which is similar to the consequent of the basic plan in Fig. 7, is for finding the minimum. Consequent 1.1 further simplifies the resulting annotations based on special values of *lhs#* and the analysis information of the control variable *v#*. Consequent 2 is for finding the maximum.

Using the tree-structured plans can lead to a reduction in the size of the knowledge base since several plans can be

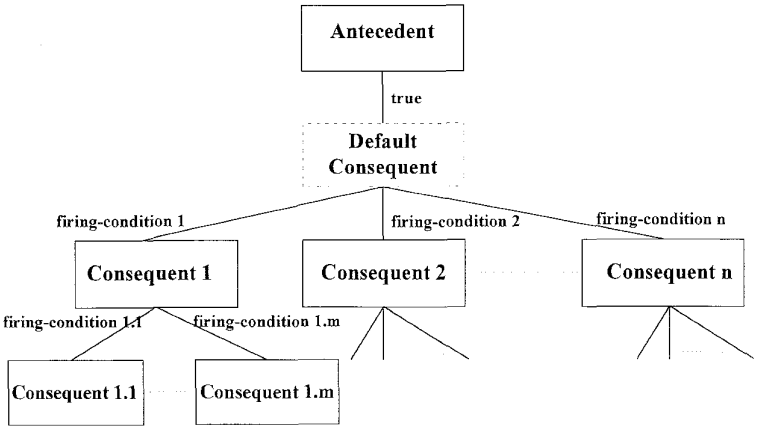

Fig. 8. The tree structure of a plan.

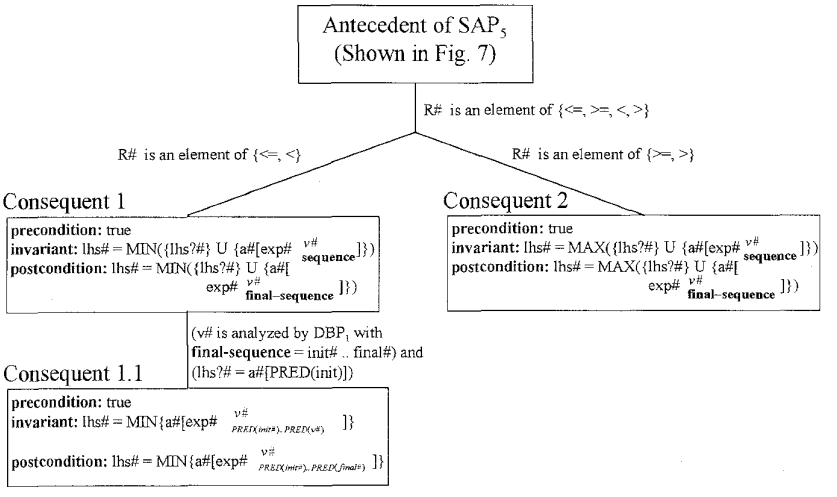

Fig. 9. Structural improvement to the plan SAP_5.

combined together into a larger one having a unique antecedent. However, the identification of the proper consequent becomes more complicated due to the required tree search.

4.5 Analysis of the Events

The events are analyzed by trying to match them with the antecedents of the knowledge base plans. 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 of the loop are formed by taking the conjunction of the corresponding parts of the event analysis results. When some event(s) do not match any library plans, the analysis only generates partial specifications of the loop.

To represent the results of matching loop events with plan antecedents, we define the *Analysis Knowledge* notation. The *Analysis Knowledge*, $AK(v)$, of a variable v modified by a certain loop event consists of an n-tuple where n is dependent on the specific matched plan. The first term of the tuple is the name of the matched plan. The remaining (n – 1) terms are the results of matching the # terms with the actual values in the event.

The resulting AK tuples for the events of the loop given in Fig. 3 are shown in Fig. 10. The first line of Fig. 10 shows that the event that modifies the variable j is matched with the antecedent of plan DBP_1 (Fig. 6). The plan variables *var#* and *var?#* are matched with the event variables j and $j?$, respectively. The plan relational operator *R#* and expression *exp#* are matched with < and *num_of_rooms* + 1, respectively. The remaining two lines of Fig. 10 can be similarly interpreted. This AK information is used to instantiate the consequents of identified plans. The instantiation results are given in Fig. 11. In this figure, the event and plan responsible for the production of each predicate are shown to its left. The first two events are analyzed by the plans DBP_1 (Fig. 6) and SAP_5 (Fig. 7), respectively. The plan, SAP_{n1}, which analyzes the third event is not shown here because it is similar to the plan SAP_5. It searches for the location of the minimum instead of the minimum. Finally, the synthesized loop specifications are shown in Fig. 12.

$AK(j) = (DBP_1, var\#: j, var?\#: j?, R\#: <, exp\#: num\_of\_rooms + 1)$
$AK(min) = (SAP_5, v\#: j, a\#: capacity, exp\#: j, lhs\#: min, lhs?\#: min?)$
$AK(index) = (SAP_{n1}, v\#: j, a\#: capacity, exp\#: j, rhs\#: min, rhs?\#: min?, lhs\#: index, lhs?\#: index?)$

Fig. 10. The AK tuples for the events of the loop given in Fig. 3.

Precondition:

Event	Plan	Predicate
1	DBP_1	$j? - 1 < num\_of\_rooms + 1$
2	SAP_5	*true*
3	SAP_{n1}	$capacity[index?] = min?$

Invariant:

Event	Plan	Predicate
1	DBP_1	$j? \le j < num\_of\_rooms + 2$
2	SAP_5	$min = MIN(\{min?\} \cup \{capacity[j? .. j - 1]\})$
3	SAP_{n1}	$capacity[index] = min$

Postcondition:

Event	Plan	Predicate
1	DBP_1	$j = num\_of\_rooms + 1$
2	SAP_5	$min = MIN(\{min?\} \cup \{capacity[j? .. num\_of\_rooms]\})$
3	SAP_{n1}	$capacity[index] = min$

Fig. 11. The instantiations for the events of the loop given in Fig. 3.

Precondition:
$(j? - 1 < num\_of\_rooms + 1)$ *and*
$(capacity[index?] = min?)$

Invariant:
$(j? \le j < num\_of\_rooms + 2)$ *and*
$(min = MIN(\{min?\} \cup \{capacity[j? .. j - 1]\}))$ *and*
$(capacity[index] = min)$

Postcondition:
$(j = num\_of\_rooms + 1)$ *and*
$(min = MIN(\{min?\} \cup \{capacity[j? .. num\_of\_rooms]\}))$ *and*
$(capacity[index] = min)$

Fig. 12. The synthesized specifications of the loop given in Fig. 3.

5 Analysis of Nested Loops

To rigorously analyze nested loops using Hoare's axiomatic approach [19], [20], the following problems need to be solved:

1) **How to represent and utilize the analysis results of inner loops?** A technique for analyzing flat loops has been described in Section 4. Can the same basic technique be used for outer loops (loops containing other loops)? What modifications, if any, need to be performed on the basic analysis technique to utilize the results of analyzing inner loops in the analysis of outer loops?
2) **How to modify the resulting specifications to facilitate Hoare-style verification**? [19], [20] This problem can be further divided into two subproblems, which are explained using the nested construct shown in Fig. 13. In this nested construct, let I_i and I_o be the invariants of the inner and outer loops, respectively.
   a) Can the above invariants be used to satisfy Hoare verification conditions that connect the specifications of inner and outer loops in the nested construct? In other words, is it possible to prove the following rules:

$$(I_i \text{ and } \neg (B_i)\ \{S_2\}\ I_o \quad (1)$$

$$(I_o \text{ and } B_o)\ \{S_1\}\ I_i \quad (2)$$

   b) If the above invariants use the notation *var?* to denote the initial value of a variable *var*, does this notation consistently refer to the value of *var* before the start of the outermost loop in the nested construct? If not, how can this inconsistency be removed?

```
while B0 do begin   Location L1
    S1
    while Bi do begin   Location L2
    end;
    S2
end;
```

Fig. 13. A nested structure of while loops.

To solve these problems, the analysis of nested loops is performed by recursively analyzing the innermost loops and replacing them with sequential constructs that represent their functional abstraction. The functional abstraction of an outer loop depends on the functional abstraction of the inner ones and not on the details of their implementation or structure.

Since this recursive analysis approach is performed bottom-up, complete knowledge of inner loop functions is available during the analysis of an outer loop. Thus, the invariant of an outer loop can be directly designed to satisfy the verification rules that are similar to rule (2) listed above. Despite the fact that inner loops are likely to contain references to variables defined in the outer loops, inner loops are analyzed in isolation of the outer ones enclosing them. As a result, a complete proof of nested constructs requires adapting the inner loop specifications to the context and initializations provided by the outer loop. More specifically, inner loop invariants and, consequently, postconditions might not be strong enough to satisfy the verification rules that are similar to rule (1). Some predicates might need to be added to the inner loop invariants and postconditions to enable the verification of such rules. The *context adaptation* phase derives these predicates and adds them to the inner loop specifications. Moreover, the consistency of using the notation *var?* to denote the initial value of a variable *var* is ensured using the *initialization adaptation* phase.

We start in Section 5.1 with some definitions that explain how we extract the initialization of a loop in a nested construct, whether it is the outermost loop or an inner one. Sections 5.2–5.4 present solutions to the two research problems mentioned above. Sections 5.2 and 5.3 offer a solution to the first research problem. Section 5.4 presents a partial solution to the second problem. In these sections, the descriptions of the analysis steps are interspersed with their application on the selection sorting example given in Fig. 14. In this example, a simple nested loop repeatedly scans an array segment searching for its minimum. It interchanges the minimum with the first element in the segment. It stops after the array *capacity*[1 .. *num_of_rooms*] has been sorted in ascending order. The inner loop of this example is the same one given in Fig. 3.

5.1 Definitions

In the following definitions, we limit the initialization of a loop to assignment statements. Conditional statements are not considered as initializations to reduce the complexity of the resulting loop specifications. Thus, resulting loop specifications are representative of the loop function without composing it with the function of preceding conditional statements.

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

Fig. 14. Example of a nested loop.

The *initialization of a loop that is not enclosed by another loop* is assumed to be a set of assignment statements of the form *identifier* := *expression*, which are immediately placed before its start. These statements give initial values for identifiers that get modified within the loop body. If this assumption cannot be satisfied or, equivalently, the loop initialization is unavailable, the notation $v?$ is used to denote the initial value of a variable v just before the start of the loop.

If we have two nested while loops, the *adaptation path* of the inner loop is a sequence of statements extracted from their control-flow graph representation. This sequence contains all the statements, simple or compound, that are completely located along the paths starting from the outer loop control predicate node and ending at the inner loop control predicate node. In this path, the relative order of the statements is kept unchanged.

The *initialization of an inner loop* in a nested construct is obtained by, first, symbolically executing its adaptation path to produce the net modification performed on each variable, if possible. Statements of the form *identifier* := *expression* are, then, extracted from the symbolic execution result. Statements are extracted if they satisfy the following two conditions:

1) The *identifier* is one of the variables modified within the inner loop body.
2) The *expression* does not reference any of the variables modified along the adaptation path.

If the initialization of a variable v that gets modified within the loop body is not given by the extracted statements, the notation $v?$ is used to denote its initial value just before the start of the loop.

The first condition, in the above definition, ensures that the initialization statements are utilized by the inner loop events. The second condition ensures that the values of *identifier* and *expression*, just before the start of the inner loop, are equal. For example, if the adaptation path is $i := i + 1; j := i$, then its symbolic execution gives the concurrent assignment $i, j := i + 1, i + 1$. Taking $j := i + 1$ as an initialization statement is not allowed because the values of j and $i + 1$, just before the start of the loop, are not equal (the values of j and i are equal). The second condition also prevents using statements of the form, say, $x := x + 1$ as initializations.

To extract the initialization of the inner loop given in Fig. 14, we use the above definitions. First, we need to symbolically execute the adaptation path of the inner loop. Since there is only one path between the start of the outer loop and the start of the inner one, the adaptation path includes all the statements completely located on this path. The adaptation path is: $index := i; min := capacity[i]; i := i + 1; j := i$.

The symbolic execution of the adaptation path yields the concurrent assignment: $index, min, i, j := i, capacity[i], i + 1, i + 1$.

Then, we need to extract initialization statements of the form *identifier* := *expression* from the symbolic execution result. The variables modified within the inner loop body are: *index*, *min*, and *j*. Thus, the statements that satisfy the first condition of the above definition are: $index := i$, $min := capacity[i]$, and $j := i + 1$. However, these statements are not valid initialization statements because their right hand sides reference the variable i that gets modified along the adaptation path. In other words, these statements do not satisfy the second condition of the above definition. As a result, the initialization statements of the inner loop are written by using the notation $v?$ to denote the initial value of a variable v as follows: $index := index?$, $min := min?$, and $j := j?$.

5.2 Analysis of Inner Loops and Representation of Their Analysis Results

The analysis of inner loops is performed using the same four phases described, in Section 4, for flat loops. To analyze an outer loop in a nested construct, the analysis results of its inner loops must be represented in a way that reveals the functionality of the inner loops and the flow of data into and out of the inner loops. The data flow information is needed to perform the decomposition of the outer loop body.

Though the resulting AK tuples or predicate logic annotations can be used to represent the inner loop analysis results, they either include too much detail or the deduction of the required information is difficult, respectively. Hence, the solution is to use a formalism that is similar to function calls; the name encapsulates the functionality while the arguments indicate the data flow information. The formalism used for this purpose is called an *Abstraction Class* (*AC*).

An AC is a knowledge base object that transforms the detailed analysis results of an inner loop to a more abstract representation that facilitates the analysis of outer loops. It groups AK tuples based on some common functionality and ignores the unnecessary implementation specific details. The

common functionality is documented to explain the purpose of designing the AC and to enhance its modifiability. Furthermore, the definition of an AC offers an abstract representation of its elements that specifies the data flow information. This abstract representation facilitates the mechanical manipulation of ACs. An Abstraction Class (AC) consists of three parts:

1) The **elements** part consists of generic AK tuples that are separated by the symbol '|'.
2) The **common-function** describes the functionality that the elements of this class share by using common instantiated **final-sequence**, **postcondition**, or **invariant** parts of the matched plans.
3) The *representation* is a unique abstract representation that gives the class name, followed by the following arguments (separated by semicolons and enclosed between two parentheses): the list of expressions responsible for the data flow into this AC, the list of variables defined by the AC, the control variables of the loop under consideration, and a unique number identifying the loop being analyzed.

The representation part contains the class name that is an arbitrary and unique name. It also contains the arguments responsible for the data flow into and out of the AC so that they can be used during the data flow analysis. The control variables and unique number of the loop are used in the design of some plan consequents. To simplify the presentation, the last two arguments are only listed when needed.

The AK of some variable belongs to a specific AC if it matches an AK tuple existing in the **elements** part. The symbol '*' is used to denote irrelevant information. An expression, *exp*, enclosed between two brackets in the **elements** part implies that the expression should be matched with the corresponding instantiated element of the actual AK to deduce the value of the variables defined in it. Some of these variables are utilized in forming the AC arguments.

The AK of the variable j analyzed in the inner loop of Fig. 14 has the following form:

$$AK(j) = (DBP_1, var\#: j, var?\#: j?, R\#: <, exp\#: num\_of\_rooms + 1).$$

This AK belongs to the AC in Fig. 15 because it matches the first AK tuple of the **elements** part. If we had implemented this loop with the condition $j \leq num\_of\_rooms$ instead of $j < num\_of\_rooms + 1$, it would have belonged to the same AC. This is because it matches the second AK tuple of the **elements** part. These two different implementations belong to the same AC because they have the common function of going over the integer sequence $j?$ .. num_of_rooms in an ascending order.

elements	$(DBP_1, var\#: [v], var?\#: *, R\#: \leq, exp\#: [final])$ | $(DBP_1, var\#: [v], var?\#: *, R\#: <, exp\#: [SUCC(final)])$
common-function	The instantiated **final-sequence** of the plan is: $v?$ .. $final$
representation	$AC_{DB_1}(v, final; v)$

Fig. 15. An abstraction class for ascending enumeration.

Using similar analysis, the AK of the variable *min* is found to belong to AC_{SA_5} (Fig. 16). The AK of the variable *index* belongs to $AC_{SA_{n1}}$. Because $AC_{SA_{n1}}$ is similar to AC_{SA_5}, it is not shown here. AC_{SA_5} includes the AK tuples that have the common function of finding the minimum of an array segment irrespective of the enumeration direction (ascending or descending) and the index of the array element being checked (v, $PRED(v)$, or $SUCC(v)$). It should be mentioned that AC_{DB_2} is similar to AC_{DB_1} but for descending enumeration. The ACs of the variables modified in the inner loop of Fig. 14 are, thus, as follows:

$$AK(j) \in AC_{DB_1}(j, num\_of\_rooms; j)$$

$$AK(min) \in AC_{SA_5}(capacity, j, num\_of\_rooms, min; min)$$

$$AK(index) \in AC_{SA_{n1}}(capacity, j, num\_of\_rooms, min, index; index)$$

elements	$(SAP_5, v\#: [v], a\#: [a], exp\#: [PRED(v)], lhs\#: [lhs], lhs?\#: *)$, where $AK(v) \in AC_{DB_2}([SUCC(final)], [SUCC(init)])$ | $(SAP_5, v\#: [v], a\#: [a], exp\#: [v], lhs\#: [lhs], lhs?\#: *)$, where $AK(v) \bullet AC_{DB_2}([final], [init])$ *or* $AK(v) \in AC_{DB_1}([init], [final])$ | $(SAP_5, v\#: [v], a\#: [a], exp\#: [SUCC(v)], lhs\#: [lhs], lhs?\#: *)$, where $AK(v) \in AC_{DB_1}([PRED(init)], [PRED(final)])$
common-function	The instantiated **postcondition** of the plan is: $lhs = MIN(\{lhs?\} \cup \{a[init .. final]\})$
representation	$AC_{SA_5}(a, init, final, lhs; lhs)$

Fig. 16. An abstraction class for finding the minimum.

After analyzing an inner loop, we replace it with the concurrent assignment that assigns to the list of variables modified by it the result of their analysis. If the AK of a variable belongs to a predefined AC, its abstract representation, as deduced from the identified AC, is assigned to it. If the AK of the variable, *var*, does not belong to a predefined AC, we assign the form *UAC*(*ak-list*; *var*) to it, where *UAC* stands for Unknown AC and *ak-list* is a list representing the AK data. The *ak-list* and *var* are used, during automatic analysis, to provide information on the unanalyzed parts of the loop.

Conceptually, the described replacement is equivalent to replacing the inner loop with a set of function calls that assign to each variable changed in the inner loop the desired value. This replacement preserves the control flow dependencies because the concurrent assignment is placed at the same relative location within the outer loop body. It also preserves the data flow dependencies between the variables because the ACs clearly state what variables are responsible for the data flow into and out of it.

Replacing the inner loops given in Fig. 14 with the described concurrent assignment gives the following modified outer loop:

```
i := 1;
while i ≤ num_of_rooms – 1 do begin
    index := i;
    min := capacity[i];
    i := i + 1;
```

$j := i$;
$j, min, index := AC_{DB_1}(j, num\_of\_rooms; j)$,
$AC_{SA_5}(capacity, j, num\_of\_rooms, min; min)$,
$AC_{SA_{n1}}(capacity, j, num\_of\_rooms, min, index; index)$;
$capacity[index] := capacity[i-1]$;
$capacity[i-1] := min$
end;

5.3 Analysis of Outer Loops

After modifying an outer loop body, we analyze it using the previously described method for analyzing flat loops (Section 4), as if it does not contain any other loops inside it. This can be done since the inner loop(s) have been replaced by ordinary sequential constructs. The only difference, in this case, is that high-level plans are used in addition to the usual (low-level) ones. High-level plans are those that utilize ACs.

Adding another classification level, based on whether the plan is low-level or high-level, to the four plan categories shown in Fig. 5, we get eight plan categories. These new plan categories are shown in Fig. 17. The advantage of this plan classification scheme is that it indexes plans for rapid access given the loop and event types.

The strength of this approach for analyzing nested constructs is that it can scale up to handle more than two nested loops. This is because the inner loops can be recursively analyzed and replaced by sequential constructs. Any outer loop can thus be analyzed by using the high-level plans in addition to the low-level ones. If we are unable to analyze one of the inner loops, the analysis of the outer loop proceeds as far as possible. That is, we can only analyze outer loop events that are independent of the unanalyzed inner loop events. In such cases, partial analysis results are produced. An outline of the application of the analysis steps on the modified outer loop of Fig. 14 is given below.

The ordered events of the modified outer loop are as follows:

1) BE (order 1)
   condition: $i \le num\_of\_rooms - 1$
   enumeration: $i := i + 1$
   initialization: $i := 1$
2) AE (order 2)
   body: $capacity[i], capacity[AC_{SA_{n1}}(capacity, i+1, num\_of\_rooms, capacity[i], i; index)] := AC_{SA_5}(capacity, i+1, num\_of\_rooms, capacity[i]; min), capacity[i]$
   initialization: $capacity := capacity?$
3) AE (order 2)
   body: $j := AC_{DB_1}(i+1, num\_of\_rooms; j)$;
   initialization: $j := j?$
4) AE (order 3)
   body: $min := (AC_{SA_5}(capacity, i+1, num\_of\_rooms, capacity[i]; min)$
   initialization: min := min?
5) AE (order 3)
   body: $index := (AC_{SA_{n1}}\ capacity, i+1, num\_of\_rooms, capacity[i], i; index)$
   initialization: index := index?

The first event is matched with the antecedent of the plan DBP_1 (Fig. 6). The second event is matched with a Simple High-level AP (SHAP) that represents the selection sorting concept. Because the variables j, *min* and *index* do not explicitly contribute to the outer loop specifications, the last three events are matched with SHAPs that produce *true* predicates. These variables implicitly affect the outer loop specifications through their abstraction classes that get used by the second event. For details concerning the plans used and the event analysis results, refer to [1]. The final synthesized analysis results are given below. The first event is responsible for the production of the first conjugate of each predicate. The second event is responsible for the production of the rest of the specifications.

Precondition:

$(0 \le num\_of\_rooms - 1)$

Invariant:

$(1 \le i \le num\_of\_rooms)$ *and*
$(FORALL\ ind: 1 \le ind \le i-1: capacity[ind] = MIN(\{capacity[ind .. num\_of\_rooms]\})$ *and*
$PERM(capacity, capacity?)$

Postcondition:

$(i = num\_of\_rooms)$ *and*
$(FORALL\ ind: 1 \le ind \le num\_of\_rooms - 1: capacity[ind] = MIN(\{capacity[ind .. num\_of\_rooms]\})$ *and*
$PERM(capacity, capacity?)$

The resulting predicate logic annotations produced for the inner and outer loops can be used to assist the understanding of the nested construct. An understanding of the sorting algorithm can be formed using the predicate

$(min = MIN(\{min?\} \cup \{capacity[j?.. num\_of\_rooms]\}))$ *and*
$(capacity[index] = min)$

of the inner loop postcondition and the predicate

$(FORALL\ ind: 1 \le ind \le num\_of\_rooms - 1: capacity[ind] = MIN(\{capacity[ind .. num\_of\_rooms]\})$ *and*
$PERM(capacity, capacity?)$

of the outer loop postcondition. However, such specifications cannot be proved using Hoare-style [19] axiomatic correctness. To be able to prove the outer loop invariant, the predicate

$(1 \le i - 1 \le num\_of\_rooms - 1)$ *and*
$(FORALL\ ind: 1 \le ind \le i-2: capacity[ind] = MIN(\{capacity[ind .. num\_of\_rooms]\})$ *and*
$PERM(capacity, capacity?)$

should be added to the invariant of the inner loop. This predicate provides information about the context of the inner loop, which is needed to prove rule (1) of the second research problem that is given at the beginning of this section. In addition, *j?*, *min?*, and *index?* in the inner loop specifications should be replaced with *i*, $capacity[i-1]$, and $i-1$, respectively.

5.4 Adaptation of Inner Loop Specifications

To be able to prove that the implementations of nested constructs satisfy their specifications, the specifications of inner

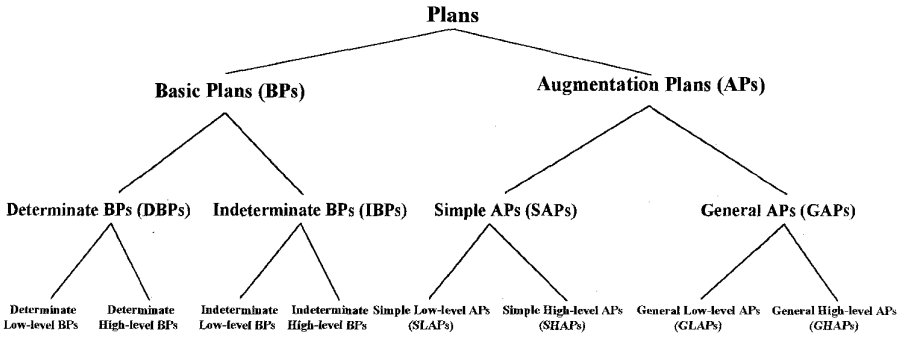

Fig. 17. New plan categories.

loops need to be strengthened to include information about the context of outer loops enclosing them. To ensure that the notation *var?* is consistently used to denote the initial value of a variable *var* before the start of the outermost loop, variables of the form *var?* in specifications of inner loops need to be replaced by their actual values. These tasks are performed in the context and initialization adaptation phases. The remainder of this subsection describes how to perform these adaptations. In this description, it is assumed that the adaptation path of an inner loop, that is defined in Section 5.1, only includes assignment and conditional statements. The cases in which this assumption is not satisfied are discussed in Section 6.

Context Adaptation. While analyzing the outer loop, we have complete knowledge of an inner loop function. Thus, this is the best time to generate a context related predicate *inner-addition*, which strengthens inner loop invariants. By studying the differences between the current outer loop **invariant** part and the generated inner loop **invariant** part, we design and add an **inner-addition** field to the consequents of the knowledge base plans. This field provides any predicates that should be added to the invariants of inner loops to enable the verification of rules similar to: $(I_i \text{ and } \neg B_i)\ \{S_2\}\ I_0$. After analyzing an outer loop, the instantiated **inner-addition** fields are synthesized, by conjunction, to form the predicate *inner-addition*.

For instance, assume that the plan DBP_1 (Fig. 6) is used to analyze an ascending enumeration construct of an outer loop having the control variable *var#*. While analyzing the inner loop in isolation, no knowledge exists about *var#* being an outer loop control variable that scans a specific sequence of values. Hence, the **inner-addition** filed of DBP_1 should provide this information in the form of the predicate: *var?# ≤ var# R# exp#*.

Analyzing the BE of the outer loop given in Fig. 14 using DPB_1 yields the following instantiated **inner-addition**:

$(1 \le i \le num\_of\_rooms - 1)$

Similarly, when the inner loop of this sorting example is analyzed in isolation, its invariant does not include any information about the sorted segment of the array *capacity*. Thus, the **inner-addition** part of the outer loop selection sorting plan should provide the following predicate:

(FORALL ind: $1 \le ind \le i - 1$: *capacity*[*ind*] = *MIN*({*capacity*[*ind .. num_of_rooms*]}) *and PERM*(*capacity, capacity?*)

By taking the conjunction of the two instantiated **inner-addition** parts, the *inner-addition* of the example given in Fig. 14 is:

$(1 \le i \le num\_of\_rooms - 1)$ *and*
(FORALL ind: $1 \le ind \le i - 1$: *capacity*[*ind*] = *MIN*({*capacity*[*ind.. num_of_rooms*]}) *and PERM*(*capacity, capacity?*)

However, the synthesized *inner-addition* is designed to be correct at a fixed reference point which is location L_1 (see Fig. 13). This is because during the design of the library plans there is no knowledge, a priori, of the statements physically located along the adaptation path. The effect of the statements along the adaptation path should be taken into consideration to get the corresponding correct predicate, $inner\text{-}addition_2$, at location L_2.

By comparing the *inner-addition* produced for the loop given in Fig. 14 to the predicate that should be added to the inner loop specifications (given at the end of Section 5.3), it is clear that they are not exactly the same. This is because the effect of the statements along the adaptation path has not been taken into consideration yet.

The context adaptation uses *inner-addition* and the adaptation path to find $inner\text{-}addition_2$.The predicate $inner\text{-}addition_2$ is deduced by reversing the effect of the statements along the adaptation path on the variables in *inner-addition* [14]. For example, if the adaptation path changes i to $i - 1$, then all the free occurrences of i in *inner-addition* are replaced by $i + 1$ to generate $inner\text{-}addition_2$.

This reversing (or inversion) is performed, mechanically, by introducing a set of auxiliary variables that replace all the free occurrences, in *inner-addition*, of the variables modified along the adaptation path. Conceptually, the auxiliary variables denote the state of the corresponding original ones at location L_1.

For the example shown in Fig. 14, the auxiliary variable i_1 replaces the variable i in *inner-addition*. Since the variable *capacity* is not modified along the adaptation path, no corresponding auxiliary variable is introduced for it. The modified *inner-addition*, which is called $inner\text{-}addition_1$, has the form:

$(1 \le i_1 \le num\_of\_rooms - 1)$ and
(FORALL ind: $1 \le ind \le i_1 - 1$: *capacity*[*ind*] = *MIN*({*capacity*[*ind.. num_of_rooms*]}) *and PERM*(*capacity, capacity?*)

We then form a predicate, *aux-values*, that represents the relation between the auxiliary variables, used at location L_1, and the corresponding original ones, used at location L_2. This predicate is formed using the symbolic execution result of the adaptation path. First, the introduced auxiliary variables should replace their corresponding actual ones that are responsible for the data flow into the symbolic execution result. The predicate equivalent of the statements that modify the original variables are, then, generated and conjunctioned together.

The predicate equivalent of an assignment statement is produced by replacing the assignment sign with an equal sign. Conditional assignments can be converted into assignment statements of the form: *var* := *choice* (*condition1*, *value1*, *condition2*, *value2*, ..., *etc.*), where the right hand side is equal to *value1* if *condition1* is *true*, *value2* if *condition2* is *true*, and so on. The resulting assignment statement is converted into a predicate as described before.

In the example shown in Fig. 14, the symbolic execution result of the adaptation path is:

index, *min*, *i*, *j* := *i*, *capacity*[*i*], *i* +1, *i* +1.

The context adaptation replaces i by i_1 in the right hand side to produce:

index, *min*, *i*, *j* := i_1, *capacity*[i_1], i_1+1, i_1+1.

The statement that modifies the original variable i is $i := i_1 + 1$. The predicate equivalent of this statement, $i = i_1 + 1$, is the predicate *aux-values*.

The required correct predicate $inner\text{-}addition_2$ is the conjunction of *aux-values* and $inner\text{-}addition_1$. The predicate $inner\text{-}addition_2$, which is actually added to the inner loop invariant, has the form:

$(1 \le i_1 \le num\_of\_rooms - 1)$ *and*
(*FORALL ind*: $1 \le ind \le i_1 - 1$: *capacity*[*ind*] =
MIN({*capacity*[*ind* .. *num_of_rooms*]}) *and*
PERM(*capacity*, *capacity?*) *and*
$i = i_1 + 1$

Initialization Adaptation. The initialization adaptation replaces each variable of the form *var?*, in an inner loop specification, with its value as deduced from its adaptation path and the invariant of the enclosing loop. After this replacement, the notation *var?* is reserved for referring to the state of a variable *var* before the start of the outermost loop. The notation var_{outer} is used to refer the value of *var* as deduced from the invariant of the loop enclosing it.

The initial value of a variable *var* is extracted from the symbolic execution result of the adaptation path. If the symbolic execution result assigns the value var_{adapt} to *var*, then var_{adapt} is the needed initial value. However, var_{adapt} needs to be modified so that it is expressed in terms of the program state at location L_2 and not location L_1. This modification is performed in the same way we modified *inner-addition*. That is, var_{adapt} is modified to $\mathrm{var}_{adapt}{}^{original\ variables}_{auxilliary\ variables}$. However, if *var* itself occurs in var_{adapt}, it should, first, be replaced by var_{outer} to avoid a circular definition of the initial value of *var*. In short, every *var?* in the inner loop specification is replaced by $((var_{adapt})^{var}_{var_{outer}})^{original\ variables}_{auxiliary\ variables}$.

For instance, the symbolic execution result of the adaptation path of the example shown in Fig. 14 is:

index, *min*, *i*, *j* := *i*, *capacity*[*i*], *i* + 1, *i* + 1.

The variable *j?* in the inner loop specification is replaced by $((i+1)^{j}_{j_{outer}})^{i}_{i_1})$, where $i = i_1 + 1$. So, *j?* is effectively replaced by *i*. Similar analysis shows that *min?* and *index?* should be replaced by *capacity*[*i* − 1] and *i* − 1, respectively.

In summary, the specification of the inner loop shown in Fig. 14 is adapted by adding the predicate $inner\text{-}addition_2$ that is simplified to:

$(1 \le i - 1 \le num\_of\_room - 1)$ and
(*FORALL ind*: $1 \le ind \le i - 2$: *capacity*[*ind*] =
MIN({*capacity*[*ind* .. *num_of_rooms*]}) *and*
PERM(*capacity*, *capacity?*)

The initial variables *j?*, *min?*, and *index?* are replaced with *i*, *capacity*[*i* − 1], and *i* − 1, respectively. These adaptation results are exactly the ones described at the end of Section 5.3.

6 Discussion

In this paper, a knowledge-based program understanding approach has been described. The resulting predicate logic annotations are unambiguous and have a sound mathematical basis that allows correctness conditions to be stated and verified, if desired. The analysis approach does not rely on real-time user-supplied information and can analyze nonadjacent loop parts.

However, there are limitations to this approach. These are:

- Practical limitations related to the effort and ingenuity needed to design the plans.
- Theoretical limitation related to the generation of concise postconditions for general loops.
- Theoretical limitation related to the adaptation of inner loop specifications in some nested loops.

The practical limits stem from the plan designers inability to formally analyze complicated loops and find their invariants despite the fact that these invariants exist theoretically. The resulting specifications are as accurate, readable, and correct as the plans are. That is why the tasks of designing plans and managing the knowledge base, for a specific application domain of interest, should be performed by an expert in both the desired domain and formal specifications.

The first theoretical limit was discussed in Section 4.4. In case of general loops, we cannot produce loop postconditions as intelligently and concisely as for simple loops because it was not possible to include postcondition parts in the plans designed for analyzing individual events of general loops. Thus, additional simplifications of the postconditions that transforms them into more readable ones might be required.

The second theoretical limit occurs in nested structures having the following characteristic: the adaptation path of an inner loop contains statements other than assignment and conditional statements (e.g., loops or procedure calls).

The context and initialization adaptations cannot, in general, be performed for such cases. The reason for this limitation is that the context and initialization adaptations are based on the fact that assignment statements and, to a lesser extent, conditional statements can be easily inverted in a mechanical way [14]. However, if there are loops, procedure calls, or function calls, this inversion cannot be performed mechanically. Performing such an inversion is equivalent to finding the specifications of arbitrary program fragments containing nonsequential constructs and representing their analysis results in terms of equational specifications that can be easily inverted. The presented approach can perform symbolic execution of sequential constructs and can produce first order predicate logic specifications of loops. However, these two different capabilities have not been integrated to produce invertible equational specifications of arbitrary program fragments.

The second theoretical limitation only affects the ability to prove that the loop implementations satisfy the resulting specifications. It does not affect the ability to assist the understanding of nested loops. This is because the approach still produces meaningful specifications of the whole construct. For instance, it has been shown that an understanding of the sorting algorithm in our example was possible before performing the adaptation steps. In addition, the context and initialization adaptation can be performed in some special cases. One special case occurs when the variables used in the *inner-addition* do not get modified along the adaptation path. Another special case happens when variables, whose initial values need to be replaced, do not get modified along the adaptation path. In the first case, the context adaptation does not need to modify the predicate *inner-addition*. In the second case, the initialization adaptation directly replaces *var?*, if any, with its value as deduced from the outer loop invariant. A third special case occurs when the loops located on the adaptation path are simple ones. In this case, the adaptation of an inner loop specification can be performed using postcondition parts of its preceding loop, which are in equational form, instead of its outer loop invariant. It should be noted that the first theoretical limit partly affects the second one. If we were able to include equational postconditions in the plans that analyze general loops, they could have been used in the adaptation steps.

7 Case Study

The program chosen as a case study of our loop analysis process deals with scheduling a set of university courses. It has about 1,400 lines of executable Pascal source code. There are a total of 39 modules (functions and procedures). A complete listing of the requirements, specifications, design, and code documents is given elsewhere [22]. In this program, there are 77 loops that cover all the classes in our taxonomy. Many of these loops involve sorting, searching, and scheduling algorithms. Because of the interactive nature of this program, it contains several other loops that perform input error detection as well as warning and error messages generation.

7.1 Objectives

The main objective of this case study was to test our analysis approach and to assess its effectiveness when applied to a fixed set of loops in a real and pre-existing program of some practical value. To this effect, we collected the data needed for performing the following validations and characterizations:

- Test the hypothesis that a loop complexity dimensions are valid indicators of its amenability to analysis.
- Test the hypothesis that the loop decomposition and plan design methods of our approach can make the plans applicable in many different loops and, hence, increase their utilization.
- Characterize the practical limits of the analysis approach.

7.2 Method

The case study was performed, manually, prior to the implementation of the prototype tool. Case study results are, thus, not affected by the limits of the implementation that are given at the end of Section 8. The set of 77 loops in the described program were extracted along with their initializations. This set included 25 *for* loops, which were transformed to their equivalent while loops. The loops analyzed had the usual programming language features such as pointers, procedure and function calls, and nested loops. To design, and prove, assertions of loops containing pointer variables, the notation and techniques of Luckham and Suzuki [30] were used. Procedures that were called from within loops had to be formally analyzed, using Hoare techniques [20], to obtain rigorous descriptions of their functionality and data flowing into and out of them.

During the study, every loop under consideration was first decomposed into its basic and augmentation events. Then, every event was analyzed in order to design a plan suitable for it. If no plan was available in the knowledge base to match the event under consideration, or a similar event, a new plan was developed with designer defined, candidate specifications. The plan was then modified and tailored to give correct specifications by trying to prove the loop invariant using Hoare techniques [19]. If a plan that matched a similar event, but not the exact one under consideration, existed in the knowledge base, improvements on the structure and/or the knowledge represented in the existing plan were considered.

As the number of analyzed loops increased, the experience gained led to the evolution of the knowledge base. The monitored usage of the knowledge base served to improve some of the plans in terms of their structure, knowledge representation, number, and naming conventions. As a result, the knowledge base was more suitable for the domain under consideration.

The designed plans (BPs and APs) were not only limited to those which provided functional specifications but also included plans that discarded unnecessary detail about temporary variables and plans that provided warning and error messages. It should also be mentioned that the resulting formal specifications were not formulated in terms of concepts specific to the application domain. Even though

such domain independent specifications can increase the chance of reusing the plans, they sometimes have the disadvantage of being more difficult to read [7].

We decided not to specifically design plans for the analysis of 12 loops (15.6%) in the case study. The unique and complex nature of these loops suggested that the effort needed to design their analysis plans highly outweighs advantages that could be gained from their expected extent of utilization in this specific application domain. That is, the partial analysis of the 12 loops in this case study is mainly attributed to the practical limitation discussed in the previous section. These 12 loops were arbitrarily numbered from p1 through p12. They were analyzed using the available set of plans to determine whether useful partial specifications could be obtained.

7.3 Results and Analysis

Tables 2 and 3 give the data collected to test the hypothesis that a loop complexity dimensions are indicators of its amenability to analysis. Table 2 gives the number of loops completely analyzed in each class defined by our taxonomy. Along the first dimension, the available and analyzed numbers of Simple (S) and General (G) loops are given. In the second dimension, the available and analyzed numbers of loops with Noncomposite (N) and Composite (C) conditions are given. Finally, the available and analyzed numbers of Flat (F) and Nested (N) loops are given along the third dimension. Using the three classification dimensions, any loop must belong to one of the 8 (2^3) equivalence classes given in Table 3. In this table, the available and analyzed numbers of loops in each of these equivalence classes are shown. The table also gives the total numbers of events and their averages for the analyzed loops in each class.

The results given in Tables 2 and 3 support the hypothesis that the classification taxonomy helps in predicting a loop amenability to analysis. Table 2 shows that the presumably more complex classes always have lower percentages of completely analyzed loops than the presumably less complex ones. For example, the percentages of completely analyzed flat and nested loops are 98 and 54, respectively. All flat loops were completely analyzed except for one loop (loop p10) that contained a call to a procedure with a partially analyzed nested loop (loop p9). This percentage variation is even more notable when further investigated along the five available equivalence classes of Table 3. Percentages range from 100% for SNF and SCF to 22% for GCN. The numbers of events in the analyzed loops further support the interpretation that the classification of a loop is an indicator of its complexity and, correspondingly, its amenability to analysis. For example, while SNF loops (Flat) have an average of 2.4 events/loop, SNN loops (Nested) have an average of 5.6 events/loop.

Table 4 summarizes the data collected to examine the plan utilization issue. It shows the number of events analyzed by each of the designed plans. It also shows the total utilization of the plans in each of the six available categories. Since only one high-level basic plan (IBP_6) was designed, we do not differentiate between low and high-level BPs. During the iterative process of designing the plans, some of them underwent abstractions and others were structurally improved into tree structures. The * or + superscript is used to denote those plans that underwent abstractions or structural improvements, respectively. For example, plan DBP_1 was used 45 times and had a tree-structured design.

The 48 plans designed were utilized in analyzing a total of 235 events. A closer examination of the results in Table 4 shows that a set of 27 plans (56%) analyzed 214 events (91%). The remaining 21 plans were only used once. These results indicate that if we focus on a specific application domain, there is bound to be a kernel of events that can be captured by a relatively reasonable number of plans. On the other hand, there will also be plans that, as in our study, may be used just once. The emphasis should be on the design of the plans that cover the kernel.

TABLE 2
NUMBER OF COMPLETELY ANALYZED LOOPS ALONG THE THREE DIMENSIONS

Analysis statistics	Dimension					
	1		2		3	
	Simple loop	General loop	Noncomposite condition	Composite condition	Flat body	Nested body
Available number	52	25	46	31	53	24
Number analyzed	48	17	42	23	52	13
Percentage analyzed	92	68	91	74	98	54

TABLE 3
NUMBER OF COMPLETELY ANALYZED LOOPS IN THE AVAILABLE CLASSES

Analysis statistics	Equivalence class							
	SNF	SCF	SNN	SCN	GNF	GCF	GNN	GCN
Available number	31	6	15	0	0	16	0	9
Number analyzed	31	6	11	—	—	15	—	2
Percentage analyzed	100	100	73	—	—	94	—	22
Number of events	75	18	61	—	—	51	—	8
Average events/loop	2.4	3	5.6	—	—	3.4	—	4

TABLE 4
UTILIZATION OF THE DESIGNED PLANS

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

The 10 plans that underwent improvements to their structure and knowledge representation (21%) analyzed 149 events (63%). The average number of utilization of the plans vary from 4.9 (with standard deviation of 8) for all 48 plans to 14.9 (with standard deviation of 11.8) for the 10

improved plans that are marked with the * and + superscripts. These numbers support the argument that commonly used plans get more chances to be revised and adapted and this, in turn, leads to their higher utilization.

We also notice, from Table 4, that even though nine SLAPs analyzed 72 events, double the number of SHAPs (19) only analyzed 39 events. This indicates that simple 'low-level' blocks of code are more frequently utilized than the more complex 'high-level' ones.

In general, the results in Table 4 show that the events/plan ratio is high (4.9), especially in case of the plans that underwent structural and knowledge representation improvements (14.9). This indicates that the decomposition and plan design methods tend to have a positive effect on plan utilization and, consequently, on the size of the knowledge base. However, since our main objective was to validate and evaluate the analysis approach, we designed many plans (21) that were only used once. These plans helped us in evaluating the analysis approach in loops with, say, high nesting level or a large number of procedure calls. Since these plans were designed to handle single specific events, they are probably not fully developed. The analysis of more loops in the same application domain should either eliminate or improve them.

Tables 5 and 6 summarize the data collected to determine which kinds of loops are more appropriately analyzed by the approach. Table 5 provides some insight into the practical limits of the approach. It gives different characteristics of the partially analyzed loops. Table 6 compares some of these characteristics to the corresponding ones of the completely analyzed loops. To provide a more detailed insight into the analyzed loops, some loop source codes are given in Appendix C.

The second theoretical limitation, described in Section 6, only occurred in loop p9. That is, the partial analysis of the 12 loops in this case study was mainly because of practical limitations. Analyzing loops p1–p6 and p8–p9 using the current set of plans yielded no partial results. Loop p10, whose characteristics are compatible with those of the completely analyzed loops, was almost completely analyzed; four out of five events were analyzed. The fifth event was not analyzed because it contained a call to a procedure with a partially analyzed nested loop (loop p9). Loops p7 and p12 yielded some minor partial analysis results. Loop p11 gave considerable partial analysis results.

It is clear from Table 5 that almost all of the partially analyzed loops are nested (11 out of 12) and contain procedure calls (10 out of 12). They have an average size of 43.2 executable source lines of code and an average of 12.4 modified variables. Table 6 shows that some of these characteristics are considerably different from the corresponding ones for the completely analyzed loops. For example, the completely analyzed loops have an average size of 10.5 executable source lines of code and an average of 3.4 modified variables. While the average number of events in the completely analyzed loops is 3.3, the partially analyzed loops have 11.9 events on the average. This case study has given us the impression that loops of up to five events were more easily analyzed than others.

However, we noticed in some loops (p7, p8, p11, and p12) that some events closely match some of the designed plans. A larger domain of study could have improved those plans or resulted in designing similar ones that can contribute more to the specifications of such loops.

Even though the results of the case study are encouraging, further experimentation is, in our opinion, needed to investigate the generality and efficiency of the presented approach with respect to various application domains. This experimentation can serve to characterize the cases in which this approach can work best.

TABLE 5
CHARACTERISTICS OF THE 12 PARTIALLY ANALYZED LOOPS

Loop #	Characteristics								
	Class	Events	Executable SLOC	Modified variables		Pointer variables	Procedure calls	Function calls	Inner loops
				control	non-control				
p1	GCN	13	48	3	10	0	5	4	1
p2	GCN	9	30	3	6	0	2	2	1
p3	GCN	13	46	3	10	0	3	2	2
p4	GCN	9	32	3	6	0	2	2	1
p5	GCN	13	49	3	10	0	4	2	2
p6	SNN	17	53	1	16	2	7	2	1
p7	SNN	20	53	1	19	4	0	1	1
p8	GCN	8	36	2	7	2	4	0	1
p9	SNN	5	29	1	4	3	0	0	2
p10	GCF	5	13	2	4	0	1	2	0
p11	GCN	12	52	3	11	4	1	4	2
p12	SNN	19	77	1	20	4	1	4	3

TABLE 6
COMPARISON BETWEEN THE COMPLETELY AND PARTIALLY ANALYZED LOOPS

Characteristics (in terms of average numbers)	Completely analyzed loops	Partially analyzed loops
Events	3.3 (SD = 2.1)	11.9 (SD = 4.8)
Executable SLOC	10.5 (SD = 8.3)	43.2 (SD = 15.7)
Modified variables	3.4 (SD = 2.5)	12.4 (SD = 4.9)

8 IMPLEMENTATION

To demonstrate the feasibility of automating our knowledge-based analysis approach, a prototype tool, which annotates loops with predicate logic annotations, has been designed [2]. LANTeRN, which stands for "Loop ANalysis Tool for Recognizing Natural-concepts," has been developed using Lisp. The input to the current version of LANTeRN 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. LANTeRN's output includes the loop classification, loop events along with names of the plans they match, individual event analysis results, and synthesized and adapted final results. Its knowledge bases contain plans and ACs from the case study. The test cases were also used from the case study. It should also be mentioned that the specifications presented in this paper were generated by LANTeRN.

In the current implementation, construction of the plans and ACs is not automated. That is, we manually populated the knowledge bases. However, no human interaction is needed during the utilization of the plans and ACs during analyzing loops. The construction of the tree-structured plans, especially in case of large knowledge bases, can be facilitated by the design of automated techniques that assist

in their acquisition and development. For instance, several knowledge base plans might have the same antecedent parts except for the **firing-conditions**. Other plans might have antecedents that represent special cases of a more general antecedent. Automatically identifying such plans and combining them into more sophisticated tree structures is an interesting topic for future study.

The first phase in the implementation is a translation phase that converts the input into a language independent form. The loop initialization and body are converted into a set of lisp function calls. The loop condition, however, is left in its predicate form. Data type information is also extracted. The translation results are stored in a global data base so that they can be easily accessed by all analysis phases. After the translation phase, the rest of the prototype can be used to analyze loops independent of the imperative programming language used.

Starting from the innermost loop(s), all input loops are recursively analyzed. If the loop body contains inner loop(s), the AK tuples of the inner loop(s) are used to search for the matching ACs in the AC knowledge base. The inner loop(s) are then replaced by a concurrent assignment in terms of the found ACs as explained in Section 5.2. After this replacement, the four main phases of the loop analysis approach are implemented by following the descriptions given in Section 4. Two kinds of simplifications are performed in LANTeRN. The simplification of arithmetic expressions is performed by converting input expressions into an internal canonical form for polynomials, manipulating them, and converting them back to their external form [34]. Predicate simplifications, however, are limited. They are performed using rule-based translation with a set of logical identities serving as rules.

Because LANTeRN was designed for the specific purpose of demonstrating that our approach can be automated, its user interface is primitive and the only structured type currently being handled is the array type. Because of the second limitation, all loops considered in our case study were analyzed by LANTeRN except for those which included pointers.

9 Conclusion

In this paper, a knowledge-based loop analysis approach has been described. This approach mechanically generates rigorous unambiguous predicate logic annotations of computer programs. It is a bottom-up analysis approach that does not rely on real-time user-supplied information that might not be available at all times (e.g., the goals a program is supposed to achieve). In addition, it enables partial recognition and analysis of stereotyped, nonadjacent program parts.

A case study was performed on a real and existing program of some practical value. This case study served to partially validate the analysis approach and to characterize its practical limits. To demonstrate the feasibility of automating our knowledge-based analysis approach, a prototype tool, which annotates loops with predicate logic annotations, has been designed and implemented [2].

The approach can assist in the maintenance and reuse 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 [5], [28]. Automation of program analysis and understanding can, thus, contribute to maintenance tools and methods and provide support for various maintenance activities. Program analysis and understanding is also crucial for code reuse since the reuser must be aware of what a code component does. Understanding reusable code components can be achieved by augmenting them with a precise and clear description of their functionality [7]. If these descriptions are in the form of formal specifications, they can be further used in generating test cases and assessing the correctness of the implementation. Automation of program understanding is needed to facilitate the quick and efficient population of a reuse repository with well documented components [4], [11].

However, when annotating complicated and large program parts, these formal specifications can become hard to read. The readability of such specifications can be enhanced if they are further abstracted. This abstraction can be performed by replacing a formal statement with another one that is formulated in terms of a more widely known and understood concept [13]. Domain abstractions can further abstract the formal specifications with concepts specific to the application domain. The domain specific replacements can be explicitly performed by producing the abstract and then the domain specific ones. Otherwise, they can be implicitly performed by designing the plans such that their consequents are directly written in terms of the domain specific terms. In the former case, the knowledge base plans are more general and can be used in several different domains. The last stage that performs the higher level abstractions can be tailored to the needs of different domains and thus enhances the portability of the system. The latter approach, however, is easier to implement mechanically but reduces the generality of the plans.

With respect to software development, predicate logic plays an important role in development of software using such languages as VDM and Z [25], [44], [51]. Since our loop analysis technique produces predicate logic annotations, it can assist such formal development methods. Our reverse engineering approach can provide assistance in the last development stage that moves from operation specifications to imperative programming language implementations. That is, the presented loop analysis technique can help in showing that proof obligations generated during the operation refinement process are satisfied. It should be noted, however, that the mathematical notations used in VDM, Z, and our plans are not the same. To transform one mathematical notation to another, simple syntactic variations need to be performed. For a detailed description of how our approach can assist in program development with VDM and Z, refer to [2].

There are some practical and theoretical limits to the presented approach. The practical limits are due to the difficulty of designing the knowledge base plans. The theoretical limits occur in nested structures with adaptation paths that contain statements other than assignment and conditional statements. They also occur while deriving the postconditions of general loops.

Future work includes extensions and improvements of the analysis approach, experimenting with the techniques in various application domains, and improvements on the prototype tool. The analysis approach needs to be expanded to perform an intelligent analysis of complete program modules that include nonalgorithmic constructs such as stacks and queues. We need to investigate the utilization of additional information and knowledge in the source code (e.g., comments, variable names) to assist in plan recognition. Performing empirical studies in various application domains can serve to address and investigate several issues related to the acquisition and development of plans and the generality and efficiency of the presented approach with respect to different application domains. Finally, the developed tool served to demonstrate that the analysis techniques can be automated [2]. For practical utilization of such a tool, it needs to be enhanced to support additional programming language features and improve the user interface.

Appendix A – Notation

$\neg$	The negation operator
$\Rightarrow$	The implication operator
$x \in y$	x is an element of y
$x \cup y$	Union of the sets x and y
*	Denotes an irrelevant information
var?	Value of *var* before an operation or a loop
var_{outer}	Value of *var* as deduced from the outer loop invariant
var_{adapt}	Value of *var* as deduced from the adaptation path of the current inner loop
$i \,..\, j$	Sequence of integers from i up to j inclusive
(*FORALL* x: *p1*: *p2*)	For all x values that satisfy *p1*, *p2* is true
and	The logical conjunction operator
B	While loop condition
MIN s	The minimum of the set (or sequence) *s*
MAX s	The maximum of the set (or sequence) *s*
or	The logical disjunction operator
P_y^x	The result of substituting y for each free occurrence of x in P
P{S}Q	If the predicate P is true before executing the first statement of the program part S, and if S terminates, then the predicate Q will be true after the execution of S is complete
PERM(a, b)	Array a is a permutation of the array b
PRED(x)	The predecessor of x
SUCC(x)	The successor of x

Appendix B – Acronyms

AC	Abstraction Class
AC_{DB}	AC that abstracts AK tuples whose first term is a DBP
AC_{SA}	AC that abstracts AK tuples whose first item is a SAP
AE	Augmentation Event
AK	Analysis Knowledge
AP	Augmentation Plan
BE	Basic Event
BP	Basic Plan
DBP	Determinate Basic Plan
GAP	General Augmentation Plan
GCF	General loop, Composite condition, Flat
GCN	General loop, Composite condition, Nested
GHAP	General High-level Augmentation Plan
GLAP	General Low-level Augmentation Plan
GNF	General loop, Noncomposite condition, Flat
GNN	General loop, Noncomposite condition, Nested
IBP	Indeterminate Basic Plan
LANTeRN	Loop ANalysis Tool for Recognizing Natural-concepts
SCF	Simple loop, Composite condition, Flat
SCN	Simple loop, Composite condition, Nested
SHAP	Simple High-level Augmentation Plan
SLAP	Simple Low-level Augmentation Plan
SNF	Simple loop, Noncomposite condition, Flat
SNN	Simple loop, Noncomposite condition, Nested
UAC	Unknown Abstraction Class

Appendix C – Example Loops

The following four figures provide a more detailed insight into the analyzed loops. The first two figures (Figs. 18 and 19) show two of the completely analyzed loops. The last two figures (Figs. 20 and 21) demonstrate two of the partially analyzed loops. These two loops were referenced as p1 and p_9, respectively.

```
i := 1;
course_i := 0;
flag := false;
while (i <= num_of_courses) and not flag do begin
   if course_no = course_no_db[i] then begin
      course_i := i;
      flag := true
   end else
      i := i + 1
end
```

Fig. 18. First example of a completely analyzed loop.

```
num_of_pref := 0;
valid_pref_list := nil;
while pref_list <> nil do begin
   num_of_pref := num_of_pref + 1;
   i := 1;
   flag := false;

   while (i <= num_of_times) and not flag do begin
      if pref_list^.p_index = time_slot_db[i] then begin
         new(temp_list);
         temp_list^.p_index := i;
         if num_of_pref = 1 then
            temp_list^.p_ptr := nil
         else
            temp_list^.p_ptr := valid_pref_list;
         valid_pref_list := temp_list;
         flag := true
      end else
         i := i + 1
   end;

   if i > num_of_times then begin
      writeln('line # ', line_no: 3, ' ** ', msgbuf);
      writeln('        no such preference in the db : ', pref_list^.p_index);
      num_of_pref := num_of_pref - 1
   end;
   pref_list := pref_list^.p_ptr
end
```

Fig. 19. Second example of a completely analyzed loop.

```
error := false;
num_of_rooms := 0;
get_next_line(1, buffer);
line_no := line_no + 1;
msgbuf := buffer;
flag := false;
while (buffer <> 'eof') and (num_of_rooms < maxrooms)and not flag do begin
   token := get_token([';', ':'], buffer);
   if token = '; ' then begin
      flag := true
   end;
   if not flag then begin
      if not chk_fmt_rm_no(token) then begin
         error := true
      end;

      {The following loop was completely analyzed.}
      i := 1;
      while i <= str_length do begin
         room_no[i] := token[i];
         i := i + 1
      end;
      token := get_token([';', ':'], buffer);
      if token = ': ' then
         token := get_token([';', ':'], buffer);
      cap := string_to_int(token);
       if not chk_range_cap(cap) then
         error := true;
      if not error then
         if not chk_dup(room_no) then begin
            num_of_rooms := num_of_rooms + 1;
            classroom_db[num_of_rooms].room_no := room_no;
            classroom_db[num_of_rooms].capacity := cap
         end else begin
            writeln('line #', line_no: 3, ' ** ', msgbuf);
            writeln('   classroom entry specified more than once :', room_no, '** ignored **')
         end;
      token := get_token([';', ':'], buffer);
      if token = '; ' then
         flag := true
      else
         if num_of_rooms <> maxrooms then begin
            get_next_line(1, buffer);
            line_no := line_no + 1;
            msgbuf := buffer
         end
   end
end;
```

Fig. 20. Partially analyzed loop number p1.

```
slot_full := true;
temp_pg_res := pg_reserve;
while temp_pg_res <> nil do begin

  {The following loop was completely analyzed.}
  temp_timeslots := temp_pg_res^.timeslots;
  while (temp_timeslots<>nil)and(temp_timeslots^.timeslot<>time) do
    temp_timeslots := temp_timeslots^.t_ptr;

  if temp_timeslots <> nil then begin
    temp_room_list := temp_timeslots^.roomlist;
    if (temp_room_list <> nil) and (temp_room_list^.r_index = room) then begin
      temp_timeslots^.roomlist := temp_timeslots^.roomlist^.r_ptr;
      if temp_timeslots^.roomlist <> nil then
      slot_full := false
    end else begin
      slot_full := false;
      if temp_room_list <> nil then begin

        {The following loop was completely analyzed.}
        flag := false;
        while not flag and (temp_room_list^.r_ptr <> nil) do
          if temp_room_list^.r_ptr^.r_index <> room then begin
            temp_room_list := temp_room_list^.r_ptr
          end else
            flag := true;

        if temp_room_list^.r_ptr <> nil then
          temp_room_list^.r_ptr := temp_room_list^.r_ptr^.r_ptr;
      end
    end
  end;
  temp_pg_res := temp_pg_res^.res_next
end;
```

Fig. 21. Partially analyzed loop number p9.

ACKNOWLEDGMENT

We thank Lionel Briand, Gianluigi Caldiera, Walcelio Melo, Carolyn Seaman and Barbara Swain for their helpful contributions to a variety of aspects presented in this paper. This research was supported in part by the Office of Naval Research under Grant No. NOOO14-87-k-0307 to the University of Maryland.

REFERENCES

[1] S.K. Abd-El-Hafiz and V.R. Basili, *A Knowledge-Based Approach to Program Understanding*. Kluwer Academic Publishers, 1995.

[2] S.K. Abd-El-Hafiz and V.R. Basili, "A Tool for Assisting the Understanding and Formal Development of Software," *Proc. Sixth Int'l Conf. Software Engineering and Knowledge Engineering*, Jurmala, Latvia, pp. 36-45, 1994.

[3] S.K. Abd-El-Hafiz and V.R. Basili, "Documenting Programs Using a Library of Tree Structured Plans," *Proc. Conf. Software Maintenance,"* Montreal, Canada, pp. 152-161, 1993.

[4] S.K. Abd-El-Hafiz, V.R. Basili, and G. Caldiera, "Towards Automated Support for Extraction of Reusable Components," *Proc. Conf. Software Maintenance*, Sorrento, Italy, pp. 212-219, 1991.

[5] A. Abran and H. Nguyenkim, "Analysis of Maintenance Work Categories Through Measurement," *Proc. Conf. Software Maintenance*, Sorrento, Italy, pp. 104-113, 1991.

[6] D. Allemang and B. Chandrasekaran, "Functional Representation and Program Debugging," *Proc. Sixth Ann. Knowledge-Based Software Engineering Conf.*, Syracuse, N.Y., pp. 167-178, 1991.

[7] V.R. Basili and S.K. Abd-El-Hafiz, "A Method for Documenting Code Components," *The J. of Systems and Software*, to appear.

[8] S.K. Basu and J. Misra, "Proving Loop Programs," *IEEE Trans. Software Engineering*, vol. 1, no. 1, pp. 76-86, 1975.

[9] K. Bertels, P. Vanneste, and C. De Backer, "A Cognitive Approach to Program Understanding," *Proc. Working Conf. Reverse Engineering*, Baltimore, Md., pp. 1-7, 1993.

[10] G. Brassard and P. Bratley, *Algorithmics: Theory Practice*. Prentice Hall, 1988.

[11] G. Caldiera and V.R. Basili, "Identifying and Qualifying Reusable Software Components," *Computer*, vol. 24, no. 2, pp. 61-70, 1991.

[12] D.D. Dunlop and V.R. Basili, "A Heuristic for Deriving Loop Functions," *IEEE Trans. Software Engineering*, vol. 10, no. 3, pp. 275-285, 1984.

[13] R.B. France and V.R. Basili, "A Pattern-Driven Approach to Code Analysis for Reuse," Tech. Report CS-TR-2802, Dept. of Computer Science, Univ. of Maryland, College Park, Md. 1991.

[14] D. Gries, *The Science of Programming*. Springer-Verlag, 1981.

[15] M.T. Harandi and J.Q. Ning, "PAT: A Knowledge-Based Program Analysis Tool," *Proc. Conf. Software Maintenance,* Phoenix, Ariz., pp. 312-318, 1988.

[16] M.T. Harandi and J.Q. Ning, "Knowledge-Based Program Analysis," *IEEE Software*, vol. 7, no. 1, pp. 74-81, 1990.

[17] J. Hartman, "Understanding Natural Programs Using Proper decomposition," *Proc. 13th Int'l Conf. Software Engineering*, pp. 62-73, Austin, Tex., 1991.

[18] P.A. Hausler, M.G. Pleszkoch, R.C. Linger, and A.R. Hevner, "Using Function Abstraction to Understand Program Behavior," *IEEE Software*, vol. 7, no. 1, pp. 55-63, 1990.

[19] C.A.R. Hoare, "An Axiomatic Basis for Computer Programming," *Comm. ACM*, vol. 12, no. 10, pp. 576-580, 583, 1969.

[20] C.A.R. Hoare, "Procedures and Parameters: An Axiomatic Approach," *Symp. Semantics of Algorithmic Languages*, pp. 102-116. Springer-Verlag, 1971.
[21] C.S. Hsieh, "Slice, Chunk, and Dataflow Anomaly as Datalog Rules," *The J. of Systems and Software, vol.* 16, no. 3, pp. 197-203, 1991.
[22] P. Jalote, *An Integrated Approach to Software Engineering*. Springer-Verlag, 1991.
[23] W.L. Johnson, *Intention-Based Diagnosis of Novice Programming Errors*, Morgan Kaufmann, 1986.
[24] W.L. Johnson and E. Soloway, "PROUST: Knowledge-Based Program Understanding," *IEEE Trans. Software Engineering*, vol. 11, no. 3, pp. 267-275, 1985.
[25] C.B. Jones, *Systematic Software Development Using VDM*. Prentice Hall Int'l, 1990.
[26] S. Katz and Z. Manna, "Logical Analysis of Programs," *Comm. ACM*, vol. 19, no. 4, pp. 188-206, 1976.
[27] K. Lano and P. Breuer, "From Programs to Z Specifications," *Z User Workshop, J.E.* Nicholls , pp. 46-70. Springer-Verlag, 1989.
[28] B.P. Leintz and E.B. Swanson, *Software Maintenance Management*. Addison-Wesley, 1980.
[29] S. Letovsky, "Program Understanding with the Lambda Calculus," *Proc. 10th Int'l Joint Conf. on AI*, pp. 512-514, 1987.
[30] D.C. Luckham and N. Suzuki, "Verification of Array, Record, and Pointer Operations in Pascal," *TOPLAS*, vol. 1, no. 2, pp. 226-244, Oct. 1979.
[31] K.B. McKeithen, J.S. Reitman, H.H. Rueter, and S.C. Hirtle, "Knowledge Organization and Skill Differences in Computer Programmers," *Cognitive Psychology*. vol. 13, pp. 307-325, 1981.
[32] H.D. Mills, V.R. Basili, J.D. Gannon, and R.G. Hamlet, *Principles of Computer Programming: A Mathematical Approach*. Allyn and Bacon, 1987.
[33] W.R. Murray, *Automatic Program Debugging for Intelligent Tutoring Systems*. Morgan Kaufmann, 1988.
[34] P. Norvig, *Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp*. Morgan Kaufmann, 1992.
[35] T.W. Pratt, "Control Computations and the Design of Loop Control Structures," *IEEE Trans. Software Engineering*, vol. 4, no. 2, pp. 81-89, 1978.
[36] A. Quilici, "A Hybrid Approach to Recognizing Programming plans," *Proc. Working Conf. Reverse Engineering*, pp. 126-133, Baltimore, Md., 1993.
[37] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice," *Proc. Seventh Int'l Joint Conf. on AI*, pp. 1,044-1,052, Aug. 1981.
[38] C. Rich and L.M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, vol. 7, no. 1, pp. 82-89, 1990.
[39] E. Rich and K. Knight, *Artificial Intelligence*. McGraw-Hill, 1991.
[40] P.G. Selfridge, R.C. Waters, and E.J. Chikofsky, "Challenges to the field of Reverse Engineering," *Proc. Working Conf. Reverse Engineering*, Baltimore, Md., pp. 144-150, 1993.
[41] E. Soloway, "Learning to Program = Learning to Construct Mechanisms and Explanations," *Comm. ACM*, vol. 29, no. 9,pp. 850-858, 1986.
[42] E. Soloway, J. Bonar, and K. Ehrlich, "Cognitive Strategies and Looping Constructs: an Empirical Study," *Comm. ACM*, vol. 26, no. 11, pp. 853-860, 1983.
[43] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. Software Engineering*, vol. 10, no. 5, 1984.
[44] J.M. Spivey, "An Introduction to Z and Formal Specifications," *Software Engineering J.*, pp. 40-50, Jan. 1989.
[45] J.M. Spivey, *The Z Notation: a Reference Manual*. Prentice Hall Int'l, 1992.
[46] M. Ward, F.W. Calliss, and M. Munro, "The Maintainer's Assistant," *Proc. Conf. Software Maintenance*, Miami, Fla., pp. 307-315, 1989.
[47] R.C. Waters, "A Method for Analyzing Loop Programs," *IEEE Trans. Software Engineering*, vol. 5, no. 3, pp. 237-247, 1979.
[48] R.C. Waters, "The Programmer's Apprentice: A Session with KBEmacs," *IEEE Trans. Software Engineering*, vol. 11, no. 11, pp. 1,296-1,320, Nov. 1985.
[49] B. Wegbreit, "The Synthesis of Loop Predicates," *Comm. ACM*, vol. 17, no. 2, pp. 102-112, 1974.
[50] L.M. Wills, "Flexible Control for Program Recognition," *Proc. the Working Conf. Reverse Engineering*, Baltimore, Md., pp. 134-143, 1993.
[51] J.C.P. Woodcock, "Structuring Specifications in Z," *Software Engineering J.*, pp. 51-66, Jan. 1989.

Salwa K. Abd-El-Hafiz received the BS degree in electrical engineering from Cairo University, Egypt, in 1986 and the MS and PhD degrees in computer science from the University of Maryland, College Park, Maryland, in 1990 and 1994, respectively. She is currently an assistant professor at the Engineering Mathematics Department, Faculty of Engineering, Cairo University, Giza, Egypt. Her primary research interests in software engineering are program understanding, the application of artificial intelligence to software analysis, and software specification. Other interests include software maintenance, reuse, and measurements. Dr. Abd-El-Hafiz is a member of the IEEE Computer Society.

Victor R. Basili ((M'83–SM'84–F'90) is a professor in the Institute for Advanced Computer Studies (UMIACS) and the Computer Science Department at the University of Maryland, College Park, Maryland, where he served as chairman for six years. He was involved in the design and development of several software projects, including the SIMPL family of programming languages. He is currently measuring and evaluating software development in industrial and government settings and has served as a consultant to many agencies and organizations, including IBM, Motorola, HP, Boeing, Xerox, NRL, NSWC, and NASA.

Dr. Basili works on the development of quantitative approaches for software management, engineering, and quality assurance, using models and metrics for improving the software development process and product. He helped found and is one of the principals of the Software Engineering Laboratory, a joint venture between the NASA Goddard Space Flight Center, the University of Maryland, and the Computer Sciences Corporation that was established in 1976. He received the first Process Improvement Achievement Award for the GRO Ada experiment in 1989 and the NASA/GSFC Productivity Improvement and Quality Enhancement Award for the Cleanroom project in 1990.

Dr. Basili has authored more than 100 journal and refereed conference papers. In 1982, he received the Outstanding Paper Award from *IEEE Transactions on Software Engineering* for his paper on the evaluation of methodologies. He has served as editor-in-chief of *IEEE Transactions on Software Engineering*; general chair of the 15th International Conference on Software Engineering held in 1993 in Baltimore, Maryland; program chair of the sixth International Conference on Software Engineering in 1982 in Japan; and general or program chair of several other conferences. He was treasurer of the IEEE Computing Society Board of Governors and a member of the Board of Directors of Verdix Corp. He serves on the editorial board of the *Journal of Systems and Software* and is an IEEE fellow. He is a member of the Engineering Excellence Council for the Xerox Corp. He serves as a co-editor-in-chief of a new journal, *Empirical Software Engineering, An International Journal*, to be published by Kluwer Academic Publishers beginning in April 1996.