Documenting Programs Using a Library of Tree Structured Plans

Salwa K. Abd-El-Hafiz Victor R. Basili
Department of Computer Science
University of Maryland
College Park, MD 20742, U.S.A.

Abstract

This paper presents an overview of a knowledge-based approach which helps in the mechanical documentation and understanding of computer programs. This approach performs mechanical annotation of loops by first decomposing them into fragments, called events. It then recognizes the high-level concepts, represented by the events, based on patterns, called plans, stored in a knowledge-base. We focus on the design and utilization of the plans and discuss how to generalize their structure. The generalized tree structure can facilitate plan recognition and reduce the size of the knowledge-base. A case study on a real program of some practical importance, containing a set of 77 loops, has been performed. Results concerning the plans designed for this case study are given.

1 Introduction

The activity of program understanding plays an important role in nearly all software related tasks. It is vital to the maintenance and reuse activities since it becomes very difficult to perform these tasks without a deep and correct understanding of the component to be maintained or reused. Furthermore, program understanding is indispensable for improving the quality of software development. This fact can be easily realized since code reviews, debugging, and some testing approaches all require programmers to read and understand programs.

In this paper, we present an overview of a knowledge-based approach which helps in the mechanical documentation and understanding of computer programs. Specifically, this approach facilitates the process of annotating loops with their functional abstractions. It mechanically identifies and derives as many parts of the loop specifications as possible. The parts that cannot be identified, due to an insufficiency in the knowledge-base, are linked to the corresponding parts of the loop to serve as a guide for the evolution of the knowledge-base in the domain under consideration.

We focus on the description of the objects, called plans, which are stored in the knowledge-base. These plans represent units of knowledge which are utilized in identifying abstract programming concepts in the code. We first give a simplified plan structure and show how it can be used in specifying loops. We then explain how to generalize this structure and discuss the effects of the new structure on the evolution of the knowledge-base.

Finally, we explain how we designed the plans for the analysis of loops in a program of some practical size and complexity. To have an idea about the size of the knowledge-base, we give the number of designed plans and the number of generalizations and abstractions performed on them.

2 Overview of our knowledge-based approach

In this section, we introduce a taxonomy that classifies loops according to their complexity level and describe the step by step process used for annotating loops with their functional abstractions. Even though we have designed analysis techniques that can be applied to the different loop classes, we only give a simplified description of how to analyze unnested loops. In this description, the focus will be on the design of the plans and their utilization.

Several knowledge-based techniques have been designed for documenting programs. In the transformational technique[13], a semantically equivalent but more abstract form of the input program is produced with the help of plans and transformation rules. Since the plans (or rules) can only transform fixed syntactic forms and cannot reduce non-adjacent program constructs, a large knowledge-base might be needed to compensate for these drawbacks. In the graph pars-

1063-6773/93 $3.00 © 1993 IEEE

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

Table 1. The three dimensions used for classifying loops.

ing technique[16], it becomes too expensive to perform an exhaustive graphical parsing of a program. This is because the number of sub-graphs is exponential and sub-graph isomorphism is NP-complete. Other techniques[6, 14] are based on heuristic methods that trade accuracy for simplicity. The graph parsing and heuristic techniques output documentations, more or less, in the form of structured English text.

In our approach, we can analyze stereotypical loop fragments which have non-adjacent parts. The resulting documentations are more formal and accurate than English text. However, we only focus our attention on the difficult task of documenting loops. Loops are decomposed into fragments, called events, based on the structural dependencies among the different loop segments. The resulting events are analyzed, using plans stored in a knowledge-base, to deduce their functional abstractions. The functional abstraction of the whole loop is then synthesized from the functional abstractions of its events. Expert knowledge and inference procedures are thus utilized in rigorously documenting computer programs.

This idea of analyzing loops by decomposing them into fragments was first introduced by Waters[19]. 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 over other approaches[5, 12]. The idea of analysis by decomposition has also been adopted by Basili and Mills in a different context[4]. They performed an experiment in trying to understand an unfamiliar program of some complexity. Their process consists of reducing the program to be understood to smaller parts and then creating in a step-by-step process the functions produced by those parts, combining them at higher and higher levels until a full specification is achieved.

In the next subsection, we start by defining some notations and giving the loop taxonomy.

2.1 A taxonomy for loops

Let the *representation of the while loop* be *while B do S*, where the condition B has no side effects and S may be any one-in one-out flow chart. A *control variable* of the while loop is a variable that exists in the condition B and gets modified in the body S.

While loops are classified along three dimensions. The first dimension focuses on the control computation 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 cases 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 that is similar to the behavior 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 which 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 does not contain any logical operators. In the *composite* case, logical operators exist and can connect the operands in a complicated and non-standard form. Along the third dimension, the complexity of the loop body varies between *flat* and *nested* loop structures. In flat loop structures, the loop body can not contain any other loop inside it which is not the case in nested structures.

2.2 The design of plans

The term 'plan' has been used in program understanding literature to denote two different things. In some literature [13, 15, 16, 18], plans are viewed as program fragments which represent an abstract programming concept or a stereotypical action sequences. Other literature[6, 7, 11, 14] views them as units of knowledge necessary for identifying abstract programming concepts.

To avoid any ambiguities, we have decided to adopt the conventions introduced by Harandi and Ning[6, 7]. That is, we use the term 'event' to refer to a fragment representing an abstract concept in a program and the term 'plan' to refer to a unit of knowledge required to identify such a concept. To analyze loops, we define two categories of loop events and two corresponding main categories of plans.

Basic Events (BE's) 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* is one clause of the loop condition. The *enumeration* is a segment responsible for the data flow into the *condition*. The *initialization* is the initialization of the variables defined in the *enumeration*.

After identifying the BE's, the Augmentation Events (AE's) are the remaining fragments of the loop body. An AE consists of two parts: the *body* and the *initialization*. The *body* is one segment of the loop body which is not responsible for the data flow into the loop condition. The *initialization* is the initialization of the variables defined in the *body*.

The plans, stored in a knowledge-base, are utilized in identifying stereotypical loop events. These plans are used as inference rules[6, 7]. Their structure is divided into two parts: the antecedent and the consequent. When a loop event matches the antecedent, the plan is fired. The instantiation of the information in the consequent represents the contribution of this plan to the loop specifications.

Corresponding to the two event categories, we have two plan categories: *Basic Plans (BP's)* and *Augmentation Plans (AP's)*. The BP's are used to analyze the BE's and the AP's are used to analyze the AE's. Plans are further classified according to the kind of loops they analyze. The plans used for analyzing simple loops contain more information than those used for analyzing general loops.

More specifically, the sequence of values scanned by the control variable of a simple loop can be easily written. This is 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 this sequence. For instance, consider the following two examples:

Example 1:

```
flag : boolean;
course_no_db: array[0..maxcourses] of integer;
i, course_no, course_i, num_of_courses : integer;
⋮
i := 1; course_i := 0; flag := false;
while (i <= num_of_courses) and (flag = false) do
begin
    if course_no = course_no_db[i] then begin
        course_i := i;
        flag := true
    end;
    i := i + 1
end
```

Example 2:

```
x, j, n : integer;
a : array[1..max] of integer;
⋮
j := 1;
while j < n + 1 do begin
    x := x + a[j];
    j := j + 1
end
```

The loop in Example 1 searches for a course number '**course_no**' in a course number data base '**course_no_db**'. If the course number is not found, the '**flag**' stays false. If found, the '**flag**' is set to true and the location is saved in '**course_i**'. In Example 2, the variable x is assigned the value $x_0 + \sum_{ind=1}^{n} a[ind]$, where x_0 denotes the initial value of x before the start of the loop.

In the simple loop of Example 2, the sequence scanned by the control variable at any point during the loop execution is 1 to $j-1$. This sequence is needed to write the following part of the invariant: $x = x_0 + \sum_{ind=1}^{j-1} a[ind]$. The final sequence of values scanned by the control variable in this loop is 1 to n. This sequence is needed to write the postcondition: $x = x_0 + \sum_{ind=1}^{n} a[ind]$.

The analysis of *general loops* is, however, 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. In the general loop of Example 1, there is no guarantee that the final sequence scanned by the control variable i will be 1 to n. The content of the final sequence is dependent on the contents of the variables '**course_no**' and '**course_no_db**'. 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.

To accommodate the differences between simple and general loops, we have two categories of BP's. Determinate BP's (DBP's) which contain in their con-

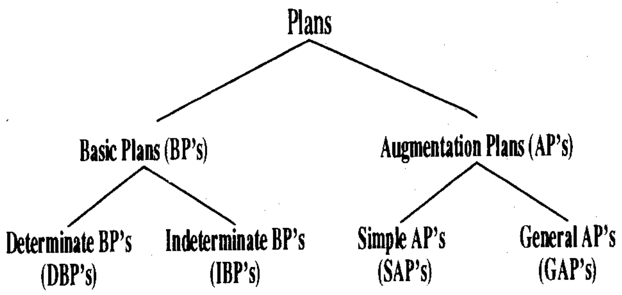

Fig. 1. Plan categories.

plan-name	DBP_1
antecedent	
control-variables	$var\#$
condition	$var\#\ R\#\ exp\#$
enumeration	$var\# := SUCC(var\#)$
initialization	$var_0\#$
firing-condition	$(R\#$ equals $\leq$ or $<) \wedge$ $(var\#$ is of a discrete ordinal type$) \wedge$ $(B$ is noncomposite$)$
consequent	
precondition	$PRED(var_0\#)\ R\#\ exp\#$
invariant	$var_0\# \leq var\#\ R\#\ SUCC(exp\#)$
postcondition	$var\# = SUCC(SHIFT(exp\#))$
seq	$var_0\# .. PRED(var\#)$
final-seq	$var_0\# .. SHIFT(exp\#)$
inner-addition	$var_0\# \leq var\#\ R\#\ exp\#$
where,	
$a..b$	A sequence of ordinal values that starts from a up to the final value b with increments of a unit step.
$SUCC$	The unary operator that gives the successor of its argument.
$PRED$	The unary operator that gives the predecessor of its argument.
$SHIFT$	The identity function if $R\#$ equals $\leq$. Equals PRED otherwise.

Fig. 2. A Determinate Basic Plan (DBP).

plan-name	SAP_1
antecedent	
control-variables	$v\#$
body	$cond\# \Longrightarrow lhs\# := lhs\#\ op\#\ exp\#$
initialization	$lhs0\#$
firing-condition	$exp\#$ does not include $lhs\#$ $\wedge$ $(lhs\# \neq v\#) \wedge$ $(op\# \in \{+, \&, *\})$
consequent	
precondition	*true*
invariant	$lhs\# = lhs0\#\ op\#\ op\#(exp\#\vert^{v\#}_{ind}$ $, ind =$ **seq**, $cond\#\vert^{v\#}_{ind})$
postcondition	$lhs\# = lhs0\#\ op\#\ op\#(exp\#\vert^{v\#}_{ind}$ $, ind =$ **final-seq**, $cond\#\vert^{v\#}_{ind})$
inner-addition	same as invariant
where,	
$op(exp_i,\ i = seq,\ cond_i)$	The result of applying the op operation to the sequence exp_i, where i varies over the sequence seq and exp_i satisfies $cond_i$.
$P\vert^x_y$	The result of substituting y for each free occurrence of x in P.

Fig. 3. A Simple Augmentation Plan (SAP).

sequents information regarding the postcondition and the sequences of values scanned by the control variable. Indeterminate BP's (IBP's), on the other hand, do not contain such information. We also have two categories of AP's. Simple AP's (SAP's) utilize the above sequences in writing the loop specifications, including its postcondition. General AP's (GAP's) do not include the loop postcondition part or utilize the above sequences. These plan categories are shown in Fig. 1. It should be noticed that if we neglect the information regarding the control sequences and the postcondition, DBP's can be used in analyzing general loops. However, the reverse is not true because DBP's are more specific than IBP's.

Figures 2-5 show four plans, one of each category. 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 are not matched (or instantiated) with actual values yet.

The plan DBP_1 (Fig. 2) 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 **seq**, **final-seq** and **postcondition** of this plan are written in a more general form that enables deducing the corresponding ones of the loop from the multiple BE's it contains. The plan SAP_1 (Fig. 3) represents a construct that performs an accumulation of successive values of an expression $exp\#$ in the variable $lhs\#$ when the condition $cond\#$ is satisfied. The plan IBP_2 (Fig. 4) represents a construct that terminates the loop execution, using the control variable var2# as a flag, after the condition cond2# is satisfied. The plan GAP_2 (Fig. 5) represents a construct that saves the value of the expression $exp\#$ in the variable $lhs\#$ when the condition $cond\#$, that causes the termination of the loop, is satisfied.

In general, the antecedent represents three kinds of knowledge.

1. Knowledge about the control variables which are used in the plan. The individual listing of these control variables, in the **control-variables** part,

plan-name	IBP_2
antecedent	
control-variables	$var1\#, var2\#$
condition	$var2\# = const2\#$
enumeration	$cond2\# \Longrightarrow var2\# := \neg const2\#$
initialization	$var2_0\#$
firing-condition	($const2\#$ equals $true$ or $false$)$\wedge$ ($cond2\#$ contains $var1\#$ but not $var2\#$)$\wedge$(The event that modifies $var1\#$ is analyzed by DBP_1) $\wedge$ (Initial value of $var1\#$ is $var1_0\#$)
consequent	
precondition	$var2_0\# = const2\#$
invariant	$(var2\# = const2\# \Longleftrightarrow (\forall\ var1_0\# \leq ind < var1\# : \neg cond2\#\|^{var1\#}_{ind}))\wedge (\forall\ var1_0\# \leq ind < PRED(var1\#) : \neg cond2\#\|^{var1\#}_{ind})$
inner-addition	same as invariant

Fig. 4. An Indeterminate Basic Plan (IBP).

serves to underscore their importance and to facilitate the readability and the comprehension of the plan.

2. Knowledge which is necessary for the recognition of stereotypical loop events. The BP's have the **condition**, **enumeration**, and **initialization** parts which represent abstractions of the corresponding three parts of stereotypical BE's. Similarly, the AP's have the parts **body** and **initialization** which represent abstractions of the corresponding two parts of stereotypical AE's.

3. Knowledge needed for the correct identification of the plans such as data types' information, whether a variable has been modified by a previous event or not, or the previous analysis knowledge of a variable. This knowledge is given in the **firing-condition**

The consequent of a library plan represents the following knowledge.

1. Knowledge necessary for the annotation of loops with their Hoare-style[9] specifications. That is why they have the **precondition**, **invariant**, and **postcondition** parts where **precondition** and **invariant** have the usual meaning[9]. 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. The

plan-name	GAP_2
antecedent	
control-variables	$var1\#, var2\#$
body	$cond\# \Longrightarrow lhs\# := exp\#$
initialization	$lhs_0\#$
firing-condition	($var2\#$ is of boolean type)$\wedge$ ($lhs\#$ does not occur in $cond\#$)$\wedge$ ($var2\# = const\#$ is a clause of B) $\wedge$ ($cond\# \neq true$)$\wedge$ (The event which modifies $var1\#$ is analyzed by DBP_1)
consequent	
precondition	$true$
invariant	$(var2\# = \neg const\# \Longrightarrow lhs\# = exp\#\|^{var1\#}_{PRED(var1\#)})\wedge (var2\# = const\# \Longrightarrow lhs\# = lhs_0\#)$
inner-addition	same as invariant

Fig. 5. A General Augmentation Plan (GAP).

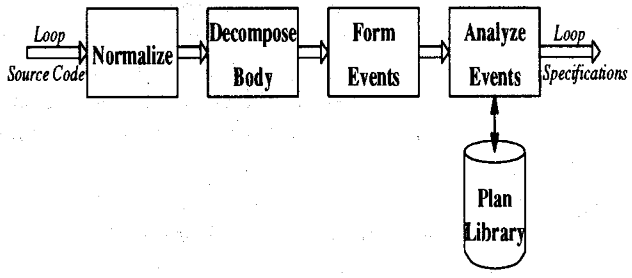

Fig. 6. Analysis of unnested loops.

inner-addition part, which is not further discussed in this paper, is needed for the complete annotation of inner loops, if any, in nested constructs.

2. In case of DBP's, knowledge about the sequence of values scanned by the control variables at any point during and after the loop execution is captured in **seq** and **final-seq** respectively.

2.3 The utilization of plans in analyzing loops

As depicted in Fig. 6, the analysis of flat loops is performed in a step by step process divided into four main phases. A brief description of each of these phases and their application to Example 1 is given in the remainder of this section.

2.3.1 Normalization of the loop representation

The purpose of this phase is to make the loop representation independent of the programming language and the implementation specific details.

The loop condition is converted into one of the well known normal forms, the *conjunctive normal form*[17]. This standard form converts a well formed formula (wff) in predicate logic into a conjunction of clauses where a clause is defined to be wff in conjunctive normal form but with no instances of the *and* ($\wedge$) connector. For example, the loop condition $x < a \vee (y < b \wedge z < c)$ is transformed to the conjunction of two clauses $(x < a \vee y < b)$ and $(x < a \vee z < c)$.

A single unwinding of the loop body is performed by symbolic execution[1, 2]. This symbolic execution makes the variables' new values dependent only on the values resulting from the last iteration of the loop, if any[3]. For example, if we have the loop body $k := t - 1; sum := sum + k$, then the symbolic execution results in the concurrent assignment $k, sum := t - 1, sum + t - 1$.

In Example 1, the condition is already in conjunctive normal form containing the two clauses $(i \leq num\_of\_courses)$ and $(flag = false)$. The symbolic execution does not affect the body of the loop. However, the net modification performed on each variable is expressed separately as follows:
$course\_no = course\_no\_db[i] \Longrightarrow course\_i := i$,
$course\_no = course\_no\_db[i] \Longrightarrow flag := true$,
$i := i + 1$

2.3.2 Decomposition of the loop body

The loop body is decomposed into minimal segments of code which are ordered according to their data flow dependencies[19]. The properties of these segments are summarized as follows:

1. Each segment gives the net modification done to some variables(s) in one iteration of the loop.
2. The set of variables defined in a segment should not be defined in any other segment.
3. Each segment, S, has a specific order. The ordering relation 'analyzed before', denoted by '$\rightarrow$', is an irreflexive partial order which is defined as follows:
   (a) If some of the variables defined in a segment S_1 are referenced in a segment S_2, then $S_1 \rightarrow S_2$.
   (b) It is possible for two segments S_1 and S_2 to be unrelated $(\neg(S_1 \rightarrow S_2$ or $S_2 \rightarrow S_1))$. Such segments are called independent.
   (c) The transitive (if $S_1 \rightarrow S_2$ and $S_2 \rightarrow S_3$, then $S_1 \rightarrow S_3$) and antisymmetric (if S_1 and S_2 are two distinct segments, then $\neg(S_1 \rightarrow S_2$ and $S_2 \rightarrow S_1))$ properties are satisfied.
   (d) Since it is meaningless for a segment S to be analyzed before itself, the irreflexive property $(\neg(S \rightarrow S))$ is satisfied.

In Example 1, the resulting segments and their orders are:

Order	Segment
1	$i := i + 1$
2	$course\_no = course\_no\_db[i] \Longrightarrow flag := true$
2	$course\_no = course\_no\_db[i] \Longrightarrow course\_i := i$

Notice that the segment that defines i has the lowest order because the other two segments reference i. The other two segments have the same order because they are independent.

2.3.3 Formation of the loop events

In this phase, we form the loop events. To form BE's, each clause of the loop condition is combined with the highest order segment(s) having data flow into it. 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 remains with no clause associated with it, its condition is set to *true*. AE's are the remaining segments of the loop body. Each event includes the initializations of the variables defined in it. By giving each event the same order as the segment it utilizes, we satisfy 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.

The loop in Example 1 contains the following three events which correspond to the three segments of the body:

1. BE (order: 1)
   condition: $(i \leq num\_of\_courses)$
   enumeration: $i = i + 1$
   initialization: $i := 1$
2. BE (order: 2)
   condition: $flag = false$
   enumeration: $course\_no = course\_no\_db[i] \Longrightarrow flag = true$
   initialization: $flag = false$
3. AE (order: 2)
   body: $course\_no = course\_no\_db[i] \Longrightarrow course\_i = i$
   initialization: $course\_i := 0$

It should be pointed out that Hausler *et.al.*[8] suggested the use of program slicing[20] to decompose loops and to allow the abstraction of loop functions one variable at time. They offered no detailed investigation or discussion of this idea. Even though the resulting loop slices are independent, each slice must include all the statements affecting the modification of the current variable. This can result in loop slices that are larger in size than our events due to the repetition of some statements in multiple slices. The plan design and identification can, in turn, be more difficult. A large knowledge-base might be needed to compensate for these problems.

2.3.4 Analysis of the events

The events are analyzed by trying to match them with the antecedents of the library plans. When an event matches the antecedent of a plan and the **firing-condition** is satisfied, the consequents of the matched plans are instantiated giving the contribution of each event to the specification of the loop. The results of these instantiations for the events of Example 1 are:

Precondition:

Event	Plan	Predicate
1	DBP_1	$0 \leq num\_of\_courses \wedge$
2	IBP_2	$false = false$

Invariant:

Event	Plan	Predicate
1	DBP_1	$1 \leq i \leq num\_of\_courses + 1 \wedge$
2	IBP_2	$(flag = false \Longleftrightarrow (\forall\ 1 \leq ind < i : course\_no \neq course\_no\_db[ind])) \wedge (\forall\ 1 \leq ind < i-1 : course\_no \neq course\_no\_db[ind]) \wedge$
3	GAP_2	$(flag = true \Longrightarrow course\_i = i-1) \wedge (flag = false \Longrightarrow course\_i = 0)$

The precondition and invariant of the loop are synthesized by taking the conjunction of the individual **preconditions** and **invariants**. The event and plan responsible for the production of each predicate are shown to its left. When some event(s) do not match any library plans, we get partial specifications of the loop.

3 Abstracting and generalizing plans

An *abstraction* of a plan is performed by replacing a formal term with another one that covers more cases. For example, replacing the addition operator, +, in a certain plan with a more abstract one which denotes either addition or multiplication represents an abstraction of this plan. The plan SAP_1 in Fig. 3 underwent this abstraction.

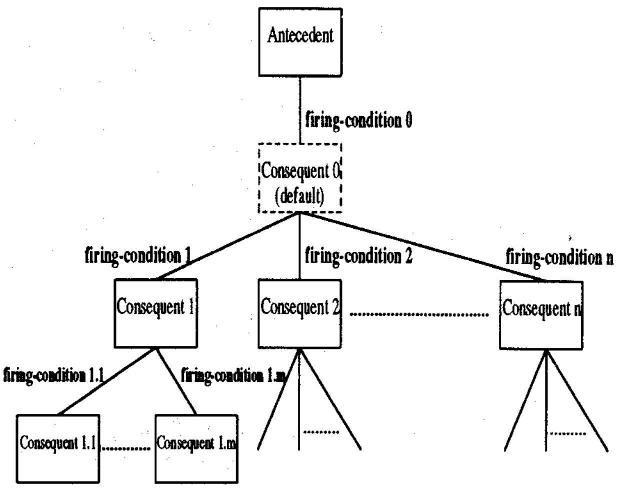

Fig. 7. The structure of a generalized plan.

A *generalization* of the plan structure into a tree structure can be performed if several similar plans are specializations of a more general case. The similar plans are grouped together in one generalized plan that has a single general antecedent and several consequents organized in one, or more, tree structures as shown in Fig. 7. The consequents are organized in one tree structure if the default consequent exists. Otherwise, they are organized in more than one tree structure (forest). In order to select a specific general plan, a match with the antecedent should occur first. Then, **firing-condition 0** must be satisfied. Within the plan, local **firing-conditions** of the consequents 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. 7 is more general than consequent 1.1). The consequents located at the same level have mutually exclusive **firing-conditions**. This means that only forward search is needed and no backtracking is required. When the event matches the antecedent and **firing-condition 0** of the generalized plan is satisfied, the search for the appropriate consequent starts at the appropriate root trying to go down in the tree as far as possible. The path between a parent and a child can only be taken if the local **firing-condition** associated with the child consequent is satisfied.

Using this generalized structure can lead to a reduction in the size of the knowledge-base since several plans can be combined together in a larger one having a unique antecedent. It can reduce the number of the plans and thus lead to a reduction in the search time needed to find a specific one. However, the instantiation of the proper consequent becomes more compli-

```
(1)  while (i <= n) and (not flag) do begin
         if condition then begin
            flag := true
              ⋮
         end else begin
              ⋮
         end
         i := i + 1;
         ⋮

(2)  while (i <= n) and (flag) do begin
         if condition then begin
            flag := false
              ⋮
         end else begin
            i := i + 1;
              ⋮
         end
         ⋮

(3)  while list <> nil and not flag do begin
         if condition then begin
            flag := true
              ⋮
         end else begin
              ⋮
         end
         list := list ↑ .pointer;
         ⋮

(4)  while list <> nil and flag do begin
         if condition then begin
            flag := false
              ⋮
         end else begin
            list := list ↑ .pointer;
              ⋮
         end
         ⋮
```

Fig. 8. Four variations of loop control computation.

cated.

Generalizations can be used to detect special cases and output loop specifications that are simpler and more concise than those for the general case. For instance, we generalized SAP_1 (Fig. 3) by making the shown consequent the default one. Three local **firing-conditions**, and their consequents, were added to detect the occurrence of special values of *exp#*, *op#* and *cond#* and to produce simplified forms of the specifications.

Considering the same loop of Example 2 with the statement that modifies x changed to '$x := x + 1$', the resulting **postcondition** using the generalized SAP_1 would be: $x_0 + n$. This form is better than the one resulting from the default consequent. The **postcondition** using the default consequent is: $x = x_0 + \sum(1,\ ind = 1..n,\ true)$.

Generalizations 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 example, IBP_2 (Fig. 4) was generalized into a plan having four consequents. The generalized plan has no default consequent. Its antecedent is similar to that shown in Fig. 4 except for the **firing-condition**. **Firing-condition 0** does not enforce a condition on the event which modifies *var1#*. Instead, four local **firing-conditions**, and their consequents, cover the four variations of loop control computation which are given in Fig. 8. The first variation is similar to the one in Example 1.

As the number of loops analyzed in a specific domain increases, the experience gained should lead to the evolution of the knowledge-base. A controlled utilization of the knowledge-base can serve to adapt some of the plans and make their abstraction level, generality, number, and naming conventions suitable for the domain under consideration. This can, in turn, facilitate their recognition and reduce the size of the knowledge-base. For example, IBP_2 underwent the described generalization and was given the more indicative name 'linear-search-IBP' during the performance of our case study.

4 A study on the design and utilization of plans

Given a fixed set of loops, the number and complexity of the plans needed to analyze them can help in judging the effect of the decomposition method and the plan design on the size of the knowledge-base.

We have chosen a program[10], having 1400 executable lines of code and 77 loops, for a study on the design and utilization of plans. This program deals with scheduling a set of courses offered by a Computer Science Department. During this case study, we had to analyze and specify loops which utilize data types such as pointers and which have a variety of Pascal statements. 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 designed with initial specifications. The plan was then modified and tailored to give correct specifications by trying to prove the loop invariant using Hoare techniques[9]. If a plan that matched a similar event, but not the exact one under consideration, existed in the knowledge-base, further abstractions and/or generalizations to the existing plan were considered.

For the 65 completely analyzed loops, containing 213 events, 48 plans were designed. Table 2 shows the plans that are utilized more than once and the number of their utilizations. The * (+) superscript is used to denote plans which underwent generalizations (abstractions) during the iterative process of their design. For example, plan DBP_1 was used 42 times and was generalized.

Since one of our objectives was to validate and evaluate our analysis techniques, we designed some plans that, in our opinion, have a very slight probability of being used more than once due to their highly specific nature. These plans helped us in evaluating the analysis techniques in loops with, say, high nesting level or a large number of procedure calls. These plans are probably not fully developed and the analysis of more loops in the same application domain should either eliminate or abstract/generalize them.

The average and standard deviation of the number of utilizations of the 48 designed plans are 4.44 and 7.25 respectively. The average and standard deviation of the number of utilizations of the 10 abstracted/generalized plans are 13.9 and 11.2 respectively. These higher numbers support the argument that commonly used plans get more chances to be revised and adapted and this, in turn, leads to utilizing them more.

More specifically, the 10 (out of 48) abstracted/generalized plans analyze 139 events out of the 213 events analyzed in this study. Those plans have a total of 24 consequents and underwent a total of 8 abstractions. This means that the experience gained during this case study enabled us to encapsulate the knowledge of at least 32 (24+8) shallow plans into 10 deep and well developed ones. This can facilitate the searching procedure and reduce the size of the knowledge-base. Coming up with such a set of generalized/abstracted plans should be the objective of any analysis performed in a specific application domain. Gaining experience in the domain should lead to the evolution of the plans into more concise and useful ones.

Name	Plan category			
(subscript)	DBP	IBP	SAP	GAP
1	42^*	4	23^{*+}	4
2	6^*	13^*	19^+	9^{*+}
3	8	2	3^*	2
4	8^*	2	10	2
5	—	2	2	2
6	—	—	3^+	—
7	—	—	13^{*+}	—
8	—	—	2	—
9	—	—	2	—
10	—	—	2	—
11	—	—	2	—
12	—	—	2	—
13	—	—	3	—

Table 2. Utilization of the designed plans.

5 Conclusion

We have presented a knowledge-based approach for specifying loops. This approach intelligently assists software engineers by adding knowledge to their environment in the form of plans stored in a knowledge-base. We have shown how to design, utilize, abstract and generalize the plans. The generalized plan structure can facilitate their recognition and reduce the size of the knowledge-base.

By performing the case study, we were able to validate our techniques and to package our experience in the design and utilization of plans for a specific domain. The issue of using the resulting specifications in a larger system that performs an intelligent analysis of complete program modules needs to be investigated. The practicality of our approach should be further investigated by trying to test them in various domains. We are in the process of designing a prototype tool which helps in performing these tasks.

Acknowledgements

Research for this study was supported in part by the ONR grant NOOO14-87-k-0307 to the University of Maryland.

References

[1] S. K. Abd-El-Hafiz, "A Tool for Understanding Programs Using Functional Specification Abstraction," Master's thesis, University of Maryland, College Park, MD 20742, 1990.

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

[3] V. R. Basili and S. K. Abd-El-Hafiz, "Packaging Reusable Components: The Specification of Programs," Technical Report CS-TR-2957, Department of Computer Science, University of Maryland, College Park, MD 20742, Sept. 1992.

[4] V. R. Basili and H. D. Mills, "Understanding and Documenting Programs," *IEEE Trans. on Software Engineering*, vol. SE-8, no. 3, May 1982, pp. 270-283.

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

[6] M. T. Harandi and J. Q. Ning, "PAT: A Knowledge-Based Program Analysis Tool," *Proc. of the Conf. on Software Maintenance*, 1990, pp. 312-318.

[7] M. T. Harandi and J. Q. Ning, " Knowledge-Based Program Analysis," *IEEE Software*, Jan. 1990, pp. 74-81.

[8] P. A. Hausler, M. G. Pleszkoch, R. C. Linger, and A. R. Hevner, "Using Function Abstraction to Understand Program Behavior," *IEEE Software*, Jan. 1990, pp. 55-63.

[9] C. A. R. Hoare, "An Axiomatic Basis for Computer Programming," *Communications of the ACM*, vol. 12, no. 10, Oct. 1969, pp. 576-580,583.

[10] P. Jalote, *An Integrated Approach to Software Engineering*, Springer-Verlag, 1991.

[11] W. L. Johnson and E. Soloway, "PROUST: Knowledge-Based Program Understanding," *IEEE Trans. on Software Engineering*, vol. SE-11, no. 3, March 1985, pp. 267-275.

[12] S. Katz and Z. Manna, "Logical Analysis of Programs," *Communications of ACM*, vol. 19, no. 4, April 1976, pp. 188-206.

[13] S. Letovsky, "Program Understanding with the Lambda Calculus," *Proceedings of the 10th Int'l Joint Conf. on AI*, August 1987, pp. 512-514.

[14] J. Q. Ning, "A Knowledge-Based Approach to Automatic Program Analysis," Ph.D. Thesis, Department of Computer Science, University of Illinois, Urbana-Champaign, Sept. 1989.

[15] C. Rich, "A Formal Representation for Plans in the Programmer's Apprentice," *Proceedings of the 7th Int'l Joint Conf. on AI*, August 1981, pp. 1044-1052.

[16] C. Rich and L. M. Wills, "Recognizing a Program's Design: A Graph-Parsing Approach," *IEEE Software*, Jan. 1990, pp. 82-89.

[17] E. Rich and K. Knight, *Artificial Intelligence*, McGraw-Hill, 1991.

[18] E. Soloway and K. Ehrlich, "Empirical Studies of Programming Knowledge," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 5, Sept. 1984 (reprinted in *Software Reusability, edited by T. J. Biggerstaff and A. J. Perlis*, vol. II, chapter 12, ACM Press, 1989).

[19] R. C. Waters, "A Method for Analyzing Loop Programs," *IEEE Trans. on Software Engineering*, vol. SE-5, no. 3, May 1979, pp. 237-247.

[20] M. Weiser, "Program Slicing," *IEEE Trans. on Software Engineering*, vol. SE-10, no. 4, July 1984, pp. 352-357.