NORTH-HOLLAND

A Method for Documenting Code Components

Victor R. Basili and Salwa K. Abd-El-Hafiz
Department of Computer Science, University of Maryland, College Park, Maryland

We propose a set of criteria for facilitating the rigorous understanding of code components via documentation and evaluate existing notations and approaches with respect to these criteria. We present an overview of an analysis approach designed to generate program documentation that satisfies these criteria. Because of the inherent difficulty and importance of reasoning about loops, we focus on understanding and documenting loops. We decompose loops into their component parts and obtain formal specifications of the resulting loop fragments by use of a knowledge base. We build this knowledge base for a specific application domain by designing plans that allow us to recognize stereotyped code patterns and associate them with their formal specifications. Finally, we synthesize a consistent and accurate specification of the whole loop construct from the specifications of its fragments. To evaluate our loop analysis approach, a case study was performed on a preexisting program of reasonable size. Results concerning the analyzed loops and the plans designed for them are given. To generate formal documentation of complete program modules, we briefly describe how to integrate our loop analysis approach with an existing program analysis tool, FSQ_2, which uses user-supplied loop specifications.

1. INTRODUCTION

Program understanding plays an important role in nearly all software-related tasks. It is vital to development, maintenance, and reuse activities. Program understanding is indispensable for improving the quality of software development. Several development activities, such as code reviews, debugging, and some testing approaches, require programmers to read and understand programs. Maintenance activities cannot be performed without a deep and correct understanding of the component to be maintained. Program understanding is vital to the reuse of code components because they cannot be used without a clear understanding of what they do. If a candidate reusable component needs to be modified, an understanding of how it is designed is also required.

Because of the importance of program understanding, considerable research has been concerned with its automation. Different automation approaches generate program documentation that assists in the understanding process and in recording the results of this understanding. Some of these approaches generate informal program documentation that gives expressive and intuitive descriptions of the code (Bertels et al., 1993; Harandi and Ning, 1990; Hartman, 1991; Johnson and Soloway, 1985; Quilici, 1993; Letovsky, 1987; Rich and Wills, 1990). However, there is no semantic basis that makes it possible to determine whether the documentation has the desired meaning. This lack of a firm semantic basis makes informal natural language documentation inherently ambiguous. Other approaches generate formal and semantically sound documentation that annotates programs according to the formal semantics of a specific model of correctness (Abd-El-Hafiz, 1990; Kemmerer and Eckmann, 1985). A common drawback of the systems that implement these approaches is that they rely on the user in the ingenious task of annotating the loops with their invariants or functions.

In this article, we describe and compare some of the formal and informal languages available for documenting code components. We argue that it is possible to produce readable abstract specifications that have an underlying sound mathematical foundation. We present an analysis approach that performs this task.

First, we briefly describe a prototype specifier that we developed to utilize user-supplied loop specifications in producing formal specifications of complete programs. However, most users find it hard to deduce such loop specifications because of the inherent reasoning difficulties involving repeated program

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

J. SYSTEMS SOFTWARE 1996; 34:89–104
© 1996 by Elsevier Science Inc.
655 Avenue of the Americas, New York, NY 10010

0164-1212/96/$15.00
SSDI 0164-1212(95)00070-H

state modifications. Because of this difficulty and the fact that loops affect the ability to understand programs (Soloway et al., 1983), we focus on the problem of generating formal loop specifications.

We introduce a knowledge-based approach to the analysis of loops. In this approach, we classify loops according to their complexity levels. Based on this taxonomy, we design the analysis techniques that best fit each of these classes. In general, we analyze loops by decomposing them into their component parts. This decomposition is based on the structural dependencies among the different loop fragments. To deduce the formal abstractions of these loop fragments, we use a knowledge base of plans. After deducing the abstractions of the loop fragments, we synthesize a consistent and accurate abstraction of the whole loop construct from the understanding of its constituents.

Finally, we describe how we designed the plans for the analysis of loops in a preexisting program of some practical size and complexity. We give the number of designed plans and show how the experience gained in the application domain affected the size of the knowledge base.

2. DOCUMENTING CODE COMPONENTS

To assist in the understanding of a code component, its documentation language should have many characteristics. Readability, expressiveness, semantic soundness, and automatability are all of particular importance (Rich and Waters, 1989). *Readability* facilitates the understanding of complicated and large code components. *Expressiveness* enables the use of the documentation language in documenting as many different code components as possible. *Semantic soundness* increases the confidence in the documentation because it allows correctness conditions to be stated and verified if desired. *Automatability* facilitates the efficient generation and manipulation of the resulting documentation.

In the remainder of this section, we discuss different possible documentation languages and techniques. In Section 2.1, a representative subset of the various documentation languages used in augmenting code components is described and evaluated in light of the aforementioned characteristics. In Section 2.2, we explain the relative strengths and weaknesses of the current documentation techniques. In our discussions, we focus on documentation used to understand the programming domain. Although application domain understanding is not discussed explicitly, the commonly occurring parts of the resulting programming domain documentation can be made to correspond to application domain primitives. The use of such application domain primitives to replace complicated terms in the documentation can improve its readability.

2.1 Alternatives for a Documentation Language

Given the wide range of documentation techniques available, the choice of a language suitable for documenting code is crucial. Documentation languages can range from free-form natural languages to abstract formal specifications. We demonstrate and compare the different choices using the bubble sort example shown in Figure 1.

Documenting programs using natural languages is an informal technique that gives an intuitive description of the code (Harandi and Ning, 1990; Rich and Wills, 1990). As pointed out by Rich and Waters (1989), the greatest strength of natural languages is their expressiveness, because they may be used to document any kind of component. However, there is no semantic basis that makes it possible to determine whether the documentation has the desired meaning. This lack of a firm semantic basis makes informal natural language documentation inherently ambiguous. With respect to readability, natural languages might appear to be easy to read just using intuition. But since they are inherently ambiguous, one must be careful not to misinterpret some statements. Because of its informal nature, judging the conciseness and clarity of natural language documentation is a difficult task that is dependent on the experience and talent of both the writer and reader. For instance, Figure 2 shows a possible English documentation of the bubble sort algorithm of Figure 1. Although the documentation in Figure 2 has some problems that are not inherent to any natural language documentation, it demonstrates that natu-

```
1    num_of_rooms: integer;
2    k, j, temp_capacity: integer;
3    capacity: array[1 .. max_rooms] of integer;

11   k := num_of_rooms - 1;
12   while k >= 1 do begin
13      j := 1;
14      while j <= k do begin
15         if capacity[j] > capacity[j+1] then begin
16            temp_capacity := capacity[j+1];
17            capacity[j+1] := capacity[j];
18            capacity[j] := temp_capacity
19         end;
20         j := j + 1
21      end;
22      k := k - 1
23   end
```

Figure 1. A bubble sort algorithm.

ral languages do not prevent the occurrence of such problems. This documentation gives a description of what is performed by the algorithm interleaved with information on how it is performed. To understand a large program, one does not usually go through the details of every part. Although information about what an algorithm does can be sufficient in some program parts, additional information on how the algorithm is designed can be useful in other parts. Thus, interleaving the two kinds of information throughout the documentation of the whole program is bound to represent an over documentation for some readers. In addition, the parts of the array *capacity* that are ordered first and the bounds of the variables j and k are not accurately stated, which might lead to misinterpretations. With respect to automatability, English text is not amenable to automatic manipulation in any significant way.

The documentation shown in Figure 3 is written in a formal specification language that uses predicate logic to produce Hoare-style annotations (Hoare, 1969). An advantage of formal specifications is that they accurately state what is performed by a program segment. Semantic soundness and expressiveness are key advantages of the predicate logic annotations. They have no trouble in representing diffuse program components and allow correctness conditions to be stated and proven if desired. Using such a mathematically sound formalism provides support for checking the consistency between the documentation and its implementation.

However, formal specifications do not give details of how a program part is designed. In case such detailed information is required, the program documentation technique should be capable of separately providing it. Moreover, when annotating complicated and large components, formal specifications become hard to read and understand. The readability of such formal 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 (France and Basili, 1991). An example of these abstractions is shown in Figure 4, which abstracts the annotations in Figure 3 by introducing the predicates

This is a bubble sort algorithm that repeatedly scans adjacent pairs of items in an array segment from one location (front) to another (end). It interchanges those items that are found to be out of order. The array *capacity*[1 .. *num_of_rooms*] is sorted in ascending order by repeatedly placing the maximum towards the end of the scanned segment.

Figure 2. English documentation of the bubble sort algorithm.

Outer loop invariant:
$num\_of\_rooms - 1 \geq k \geq 0 \wedge perm(capacity, capacity_0) \wedge$
$\forall\, k+2 \leq ind \leq num\_of\_rooms$: $capacity[ind] \geq capacity[1 .. ind - 1]$

Inner loop invariant:
$1 \leq j \leq k+1 \wedge 1 \leq k \leq num\_of\_rooms - 1 \wedge$
$capacity[j] \geq capacity[1 .. j-1] \wedge perm(capacity, capacity_0) \wedge$
$\forall\, k+2 \leq ind \leq num\_of\_rooms$: $capacity[ind] \geq capacity[1 .. ind - 1]$

where,
$perm(A1, A2)$ asserts that $A1$ is a permutation of $A2$.
var_0 denotes the initial value of a variable var just before the start of the loop.

Figure 3. Hoare-style specification of the bubble sort algorithm.

$maximum\_at\_edge(capacity[1 .. j - 1], j)$ and $upsorted(capacity[1 .. num\_of\_rooms], k + 2)$ to replace $capacity[j] \geq capacity[1 .. j - 1]$ and $\forall k + 2 \leq ind \leq num\_of\_rooms$: $capacity[ind] \geq capacity[1 .. ind - 1]$, respectively.

Domain abstractions can further abstract the formal annotations with concepts specific to the application domain. The annotation of programs with their assertions using predicate logic, algebraic specifications, and λ abstractions are examples of formal specifications. When these specifications use notations that are dependent on the application domain, for example, using $largest\_room\_at\_edge$ and $upsorted\_rooms$ instead of $maximum\_at\_edge$ and $upsorted$, respectively, they can become more understandable in their domain.

The automatic generation of specifications similar to those shown in Figures 3 and 4 is a well-known problem for the current specifiers and provers (Abd-El-Hafiz, 1990; Kemmerer and Eckmann, 1985; Good, 1985). These systems require the user to provide the loop annotations, an ingenious task that requires expert knowledge. On the other hand, the automatic manipulation of formal specifications is easier than that of informal documentation. This can be mainly attributed to the well-defined syntax and semantics of formal specifications.

In summary, formal specification languages can satisfy the aforementioned four characteristics if the

Outer loop invariant:
$num\_of\_rooms - 1 \geq k \geq 0 \wedge$
$perm(capacity, capacity_0) \wedge$
$upsorted(capacity[1 .. num\_of\_rooms], k + 2)$

Inner loop invariant:
$1 \leq j \leq k+1 \wedge 1 \leq k \leq num\_of\_rooms - 1 \wedge$
$maximum\_at\_edge(capacity[1 .. j - 1], j) \wedge$
$perm(capacity, capacity_0) \wedge$
$upsorted(capacity[1 .. num\_of\_rooms], k+2)$

Figure 4. Abstract specification of the bubble sort algorithm.

documentation technique has three features. The first feature is to separately provide, when desired, information on how some program part is designed. The second feature is allowing the use of well-known and/or domain-specific abstract terms to improve the documentation readability. The third feature is the ability to automatically generate loop annotations. On the other hand, it is very difficult to overcome the inherent ambiguity and lack of semantic soundness of informal documentation languages.

In the next subsection, we review the current documentation techniques and explain why further research is needed to produce documentation that best assists in the understanding of code components.

2.2 Alternatives for a Documentation Technique

To automatically generate documentation that facilitates the understanding of computer programs, various analysis approaches have been developed. These approaches can be classified into two broad categories: knowledge-based approaches and algorithmic approaches.

Knowledge-based approaches modularize experts' knowledge in the form of plans that can be accessed mechanically. In these approaches, the generation of a component's documentation usually involves two main tasks: the recognition of stereotyped parts in the program (Soloway and Ehrlich, 1984) and deriving their annotations using the plans stored in the knowledge base. Within these knowledge-based approaches, several analysis techniques are adopted. The transformational technique is similar to the transformational paradigm of automatic program synthesis but with the application direction of the transformation rule reversed (Letovsky, 1987; Ward et al., 1989). The problem with this technique is that it either cannot analyze nonadjacent program constructs (Letovsky, 1987) or it requires the user to choose the fragments to be analyzed and, in some cases, the transformation rules to be applied to them (Ward et al., 1989). In the graph-parsing technique (Rich and Wills, 1990), it becomes too expensive to perform an exhaustive graphical parsing of a program. This is because the number of subgraphs is exponential, and subgraph isomorphism is, in general, NP complete (Brassard and Bratley, 1988). Other techniques (Harandi and Ning, 1990) are based on heuristic methods that trade accuracy for simplicity. Both the graph-parsing and heuristic techniques output documentation more or less in the form of structured English text, which docs not have a sound semantic basis.

The algorithmic approaches, on the other hand, usually annotate programs according to the formal semantics of a specific model of correctness (Abd-El-Hafiz, 1990; Kemmerer and Eckmann, 1985). By utilizing user-provided loop annotations, they offer mechanical assistance in proving the correctness of these annotations and in producing the specifications of a complete program.

In the next section, we present a hybrid approach for mechanical generation of rigorous program documentation. It combines and builds on the strengths of both the knowledge-based and algorithmic approaches. In addition to using expert-designed plans to automatically generate intelligent analysis results, it produces formal and unambiguous specifications.

3. A HYBRID ANALYSIS APPROACH

To generate a documentation language that has the four characteristics mentioned in the previous section, our approach uses the recent advances in knowledge-based program understanding research to enhance the formal algorithmic approaches. Knowledge-based approaches are used to solve the problem of automatic generation of loop invariants or functions. This problem is regarded as formidable in algorithmic approaches. A practical program decomposition method (Waters, 1979) is used to facilitate the annotation of loops with their invariants as well as to provide insight into how they are designed. By combining the use of algorithmic and knowledge-based approaches, we satisfy the expressiveness, semantic soundness, and automatability characteristics. We improve the readability of the resulting formal specifications by replacing complicated formal statements with ones that are formulated in terms of more widely known or domain-specific concepts.

We have developed a prototype specifier that supports the derivation of programs' specifications. It is the second in a series of prototype tools developed under the general name FSQ (functional specification qualifier; Abd-El-Hafiz, 1990; Abd-El-Hafiz et al., 1991; Qian, 1989). By utilizing user-supplied loop annotations, FSQ_2 uses an algorithmic approach to analyze complete programs. In a typical session, a user derives the formal specification of a program using stepwise abstractions. The user starts by providing trial specifications of every loop in the program as a separate entity. Then, FSQ_2 assists the user in verifying whether or not the lops meet those trial specifications. After finding the actual specifications of all the loops, the correct specification of the whole program is automatically found. This

method of stepwise abstraction enables the software engineer to concentrate on small pieces of code, one at a time, and to mitigate in this way the difficulty of specifying the whole program. However, this prototype does not provide any assistance in a major and difficult task: annotating the loops with their functions or invariants. To intelligently assist in the understanding of computer programs, a technique that can enhance a specification tool, such as FSQ_2, by mechanically annotating loops is needed.

Substantial research has been performed on the specific topic of analyzing loops. The heuristic loop analysis methods can be used to guide the search for an invariant. The research performed by Dunlop and Basili (1984); Katz and Manna (1976), Remmers (1984), and Wegbreit (1974) is representative of these heuristic approaches. Although these heuristic approaches can be helpful in some cases, they can give misleading results. After applying them for a considerable amount of time, one may or may not succeed in finding a correct invariant. Other works focus on developing algorithmic approaches for finding the invariants (functions) of simple loops. Examples of the latter approaches can be found in the work of Basu and Misra (1975), Dunlop and Basili (1982), Katz and Manna (1976), Mills (1975), Misra (1978), and Morris and Wegbreit (1977).

A more practical approach, which analyzes loops by decomposing them into fragments, was proposed by Waters (1979). The key feature of this analysis method is that it uses control and data flow information to break the loop apart in a mechanical way. This decomposition can facilitate both understanding and the correctness analysis process. 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.

Hausler et al. (1990) suggested the use of program slicing to decompose loops and allow the abstraction of loop functions one variable at a 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 large in size because of the repetition of some statements in multiple slices. The identification of stereotyped slices and their analysis can, in turn, be difficult, and a large knowledge base might be needed to compensate for this problem.

It should be noted that all the loop analysis approaches mentioned earlier use formal, semantically sound, and unambiguous notation. Although they provide guidelines on how to mechanically generate loop invariants or functions, they were not actually used to implement automatic analysis systems. Consequently, we present in the next subsection a knowledge-based loop analysis approach that mechanically annotates loops with their abstract specifications.

3.1 A Knowledge-Based Approach to Loop Analysis

In this subsection, we introduce a technique based on the idea of analyzing programs by decomposition

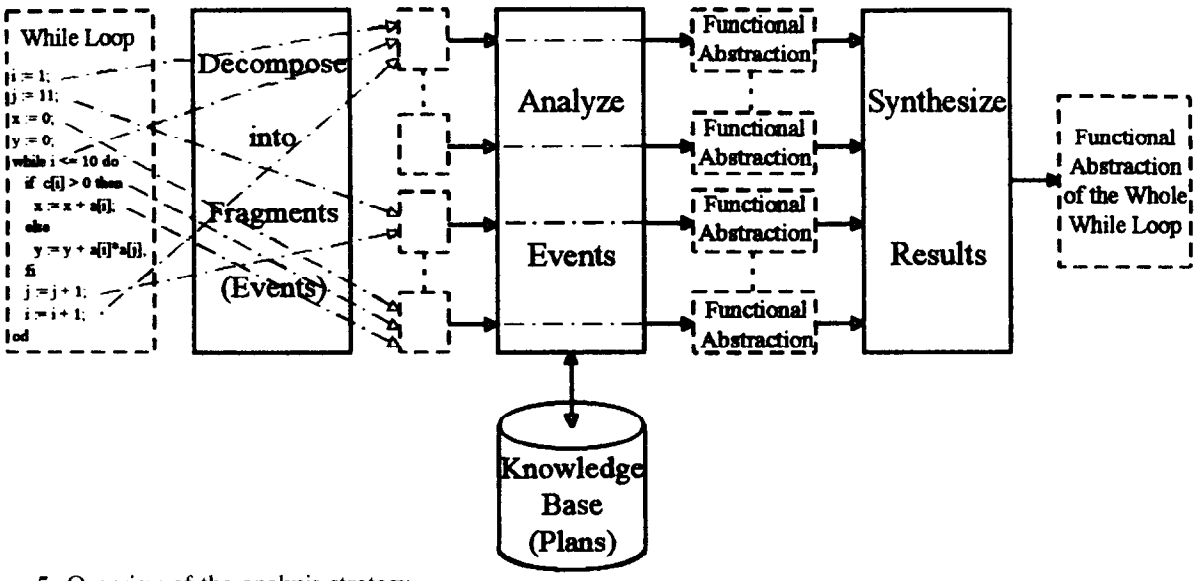

Figure 5. Overview of the analysis strategy.

(Basili and Mills, 1982; Hartman, 1991; Hausler et al., 1990; Waters, 1979). It annotates loops with their functional abstractions in a step-by-step process as depicted in Figure 5. The analysis of a loop starts by decomposing it into fragments. This decomposition is based on the structural dependencies among the different loop parts. The resulting fragments 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 fragments.

We start by introducing the notation used throughout the rest of this section.

Definition 1. 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. Even though the approach described here applies to all loops having this abstract representation, examples and illustrations are given using PASCAL.

Definition 2. A *control variable* of the while loop is a variable that exists in the condition B and gets modified in the body S.

Definition 3. A *concurrent assignment* is a statement in which several variables can be assigned simultaneously. It has a list of variables at the left-hand side of an assignment operator and an equally long list of expressions at its righthand side (i.e., $v_1, v_2, \ldots, v_n := e_1, e_2, \ldots, e_n$). Every ith expression from the righthand list is assigned to its corresponding ith variable from the lefthand list (Gries, 1981; Mills et al., 1987).

Definition 4. A *conditional assignment* is a set of one or more guarded concurrent assignments separated by commas ",". Every guarded concurrent assignment has a Boolean expression as an antecedent of an implication sign and a concurrent assignment as its consequent (i.e., $b \Rightarrow s$) (Abd-El-Hafiz et al., 1991; Gries, 1981). When the Boolean expression b is satisfied, the modifications performed on a variable are given by the concurrent assignment s. Similar to Gries' definition of the alternative command, all the guards must be well defined (Gries, 1981). However, it is possible that none of the guards evaluates to true. In this case, no variable is modified [i.e., the conditional assignment evaluates to a *skip* command (Gries, 1981)]. It should also be noted that because we are only analyzing deterministic programs, all the guards are mutually exclusive.

Definition 5. Any variable assigned in a conditional assignment defines the *data flow out* of the statement.

Definition 6. Any variable referenced by a conditional assignment defines the *data flow into* the statement.

3.1.1 Classification of loops. To design the analysis techniques that best fit different levels of program complexity, we classify the *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.

Along the first dimension, we differentiate between simple and general loops. We get a *simple loop* 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.

Because the control computation of simple loops is isolated from the rest of the loop, the sequence of values assumed by the control variable can be easily written. This is because the loop condition, the control variable's initial value, and the net modification done to the control variable in one loop iteration, if any, provide sufficient information for writing this sequence. This results in a definite behavior that is similar to the behavior of *for* loops. Consequently, we can represent their analysis knowledge with compatible definiteness and specificity.

However, this is not the case in *general loops*. In many cases, the control computation part is not isolated from the rest of the loop. For example, while performing a binary search on a sorted array, the modifications of the control variables are dependent on the content of the array segment under

Table 1. The Three Dimensions Used for Classifying Loops

Dimension	Complementary Classes	
Control computation	Simple loop	General loop
Complexity of condition	Noncomposite condition	Composite condition
Complexity of body	Flat loop	Nested loop

consideration and the value being searched for. Thus, it might not be easy, or possible, to write the sequence of values assumed by the control variables.

Simple loops cover more iteration constructs than those covered by PASCAL *for* loops. The conditions imposed on the set of PASCAL *for* loops, F, are stronger because they restrict the type of the control variable, and the control variable is only decremented or incremented by a unit step (Jensen and Wirth, 1985). That is why the set of simple loops, W_1, is a proper superset of F, that is, $F \subset W_1$. On the other hand, if W denotes the set of PASCAL *while* loops, then the set W_1 is a proper subset of W by definition. That is, $W_1 \subset W$. This is because simple loops are defined by imposing some restrictions on *while* loops.

Within the second dimension, the complexity of the loop condition B can vary between two cases. In the *noncomposite* case, B consists of only one clause of the conjunctive normal form (Rich and Knight, 1991). In the *composite* case, B consists of more than one clause of the conjunctive normal 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 cannot contain any other loop inside it, which is not the case in nested structures.

To analyze these loop classes, we have designed formalisms for representing the program knowledge as well as the plan knowledge. We have also designed analysis techniques that can be applied to these different loop classes in many domains.

3.1.2 Representation of program knowledge. Loops are decomposed into smaller meaningful parts that represent the knowledge obtained from the program text. These parts are divided into two categories, namely, basic events and augmentation events. While basic events are the parts that constitute the control computation of the loop, augmentation events are the remaining building blocks of the loop body.

Definition 7. A *basic event* (BE) consists of three parts: the condition, the enumeration, and the initialization.

1. The *condition* is one clause in the conjunctive normal form (Rich and Knight, 1991) representation of the loop condition.
2. The *enumeration* is a set of conditional assignments that assign to one or more of the control variables used in the condition the net modification done to them in one loop cycle, if any.
3. The *initialization* is a set of conditional assignments that assign initial values to the control variables modified in the enumeration part.

Definition 8. An *augmentation event* (AE) consists of two parts: the body and the initialization.

1. The *body* is a set of conditional assignments that assign the per-cycle modification taking place in the loop body to some variables other than the control variables.
2. The *initialization* is a set of conditional assignments that assign initial values to the variables modified in the body.

3.1.3 Representation of plan knowledge. The information stored in the plan knowledge base is divided into two main categories: *basic plans* (BPs) and *augmentation plans* (APs). The BPs are used to analyze the parts of the loop that control its execution, that is, BEs. The APs are used to analyze the other parts in the loop body, that is AEs.

The plans correspond to the usual rules used in a rule-based system (Hayes-Roth, 1985). They can be considered as inference rules. When a loop event satisfies a unique plan antecedent, the rule is fired. The instantiation of the information in the consequent represents the contribution of this plan to the loop assertions (Abd-El-Hafiz and Basili, 1993).

In general, an antecedent of a knowledge base plan represents three kinds of knowledge (see Figures 6 and 7):

1. The list of control variables required for the design of the plans' consequents. This list, which is maintained in the **control-variables** part, also

plan-name	DBP_1 (upward-enumeration)
antecedent	
control-variables	*var#*
initialization	$var := var_0\#$
condition	*var# R# exp#*
enumeration	*var# := SUCC(var#)*
firing-condition	*R#* is relational operator that equals ≤ or < ∧ *var#* is of a discrete ordinal type ∧ Noncomposite loop condition
consequent	
precondition	$PRED(var_0\#)$ *R# exp#*
invariant	$var_0\# \leq$ *var# R# SUCC(exp#)*
postcondition	*var# = SUCC(SHIFT(exp#))*
sequence	$var_0\#$ .. *PRED(var#)*
final-sequence	$var_0\#$ .. *SHIFT(exp#)*
where,	
a .. b	A sequence of ordinal values starting from *a* up to *b* with increments of a unit step.
SUCC (x)	The successor of *x*.
PRED (x)	The predecessor of *x*.
SHIFT	The identity function if *R#* equals ≤. Equals *PRED* otherwise.

Figure 6. Example of a basic plan (BP).

plan-name	$SLAP_1$ (maximum-at-edge)
antecedent	
control-variables	*var#*
body	*array#*[*exp#*] < *array*[*FUNC#*(*exp#*)] ⇒ *array#*[*exp#*], *array#*[*FUNC#*(*exp#*)] := *array#*[*FUNC#*(*exp#*)], *array#*[*exp#*]
initialization	—
firing-condition	(*var#* is analyzed by DBP_1 ∧ *FUNC#* = *PRED*) ∨ (*var#* is analyzed by DBP_2, ∧ *FUNC#* = *SUCC*)
consequent	
precondition	*true*
invariant	*perm*(*array#*, $array_0$#) ∧ *maximum_at_edge*(*array#*[(*FUNC#*(*exp#*)) | $^{var\#}_{sequence}$], *exp#* | $^{var\#}_{last\ (sequence)}$)
postcondition	*perm*(*array#*, $array_0$#) ∧ *maximum_at_edge*(*array#*[(*FUNC#*(*exp#*)) | $^{var\#}_{final\text{-}sequence}$], *exp#* | $^{var\#}_{last\ (final\text{-}sequence)}$)
where,	
—	Denotes irrelevant information.
$P\|^{x}_{y}$	The result of substituting *y* for each free occurrence of *x* in *P*.
sequence, final-sequence	The sequences assumed by the control variable *var#* as deduced from the analysis of its BE.
last(x)	The last element of the sequence *x*.

Figure 7. Example of an augmentation plan (AP).

serves to facilitate the readability and the comprehension of the plans.

2. Knowledge necessary for the recognition of stereotyped loop events. The BPs have the **condition**, **enumeration**, and **initialization** parts representing abstractions of the corresponding three parts of stereotyped BEs. Similarly, the APs have the parts **body** and **initialization** representing abstractions of the corresponding parts of stereotyped AEs.
3. Knowledge needed for the correct identification of the plans such as data type information, whether or not a variable has been modified by a previous event, or the previous analysis knowledge of a variable. This knowledge is given in the **firing-condition**.

A consequent of a knowledge base plan represents the following knowledge:

1. Knowledge necessary for the annotation of loops with their Hoare-style (Hoare, 1969) specifications. This is maintained in **precondition**, **invariant**, and **postcondition** parts, where **precondition** and **invariant** have the usual meaning (Hoare, 1969). The **postcondition** is only included in case of plans that analyze simple loops. It gives information 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, then no variables get modified.
2. In case of a simple loop, the constraint imposed on the control variable results in a definite behavior of the loop control computation similar to that of *for* loops. This definite behavior is captured, in the **sequence** and **final-sequence** parts of the BPs, to produce specification with compatible definiteness. These parts give knowledge about the sequence of values assumed by the control variables during and after the loop execution, respectively.

Figures 6 and 7 show two plans: a BP (DBP_1) and an AP ($SLAP_1$), 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.

The plan DBP_1 (Figure 6) represents an enumeration construct that generates a sequence of values of a discrete ordinal type in an ascending order with a unit step. The **initialization** indicates that the initial value of the unique control variable *var#*, just before the start of the loop, is var_0#. Because the **firing-condition** ensures that the relational operator *R#* equals ≤ or <, the **condition** means that the final value assumed by *var#* is determined by the expression exp#. The **enumeration** states that *var#* is incremented by a unit step. Incrementing *var#* is possible because the **firing-condition** ensures that it is of a discrete ordinal type. In the consequent, *SUCC*(*x*) and *PRED*(*x*) are defined to be the successor and predecessor of *x*, respectively. The **precondition**, **invariant**, and **postcondition** give the direct contribution of this plan to the loop specification. They assert the following: if var_0# ≤ (or <) *SUCC*(*exp#*) is true when the loop starts, then var_0# ≤ var# ≤ (or <) *SUCC*(*exp#*) remains true through successive iterations of the loop, and *var#* = *SUCC*(*exp#*) (or *exp#*) is true when the loop terminates. Because the loop condition is noncomposite, **sequence** and **final-sequence** give the values assumed by the control variable during and after the loop execution. These values are var_0#..*PRED*(*var#*) and var_0#..*exp#* [or *PRED*(*exp#*)], respectively. By saving the values of **sequence** and **final-sequence** for each simple loop under consideration and using them in the design of the augmentation plans consequents, they contribute indirectly to the loop specification.

The plan $SLAP_1$ (Figure 7) swaps successive elements of an array segment, if needed, so that the maximum element is located at one of its edges. Depending on which clause of the **firing-condition** is satisfied, the maximum is located at either the start or the end of the array segment. If the control variable *var#* is analyzed by DBP_1, the array seg-

ment is scanned in ascending order. In this case, *FUNC#* is the predecessor function *PRED*. Consequently, the **body** means that if an array element *array#*[*exp#*] is less than its predecessor *array#*[*PRED*(*exp#*)], then they are swapped so that the maximum is always located at the end. If the control variable *var#* is analyzed by DBP_2, then the array segment is scanned in descending order. In this case, *FUNC#* is the successor function *SUCC*. Consequently, the **body** means that if an array element *array#*[*exp#*] is less than its successor *array#*[*SUCC*(*exp#*)], then they are swapped so that the maximum is always located at the start. The **invariant** and **postcondition** assert these facts by using two predicates. The predicate *perm*(x, y) asserts that the array x is a permutation of the array y. The predicate *maximum_at_edge*(x, i) asserts that the maximum of the array segment x is located at the edge specified by the index i. For instance, the first predicate of the **postcondition** *perm*(*array#*, $array_0$*#*) asserts that when the loop terminates, the array *array#* is a permutation of the initial array, $array_0$*#*, at the start of the loop. The second predicate *maximum_at_edge*(*array#*[(*FUNC#*(*exp#*))$|^{var\#}_{\text{final-sequence}}$], *exp#*$|^{var\#}_{last\ (\text{final-sequence})}$) has as the first argument the array segment, which is scanned by the successive loop iterations. It is obtained by substituting every free occurrence of the control variable *var#* in the expression *array#*[(*FUNC#*(*exp#*)] with the **final-sequence** of the simple loop under consideration. The second argument specifies the edge that holds the maximum. It is obtained by substituting every free occurrence of the control variable *var#* in the expression *exp#* with the last element of the **final-sequence**.

When performing analysis of loops in a large domain, the size of the knowledge base becomes an important issue. A large increase in the number of plans leads to a large increase in the knowledge base size. To reduce the number of plans in such cases, improvements on their structure and/or the knowledge represented in them can be performed.

Knowledge representation improvements, called *abstractions*, involve replacing some of the terms in a plan with more abstract ones that cover more cases. For example, the plan $SLAP_1$ in Figure 7 can be abstracted by handling arrays whose elements can be of the record type.

Structural improvements to a plan modify the basic structure into a tree structure, which allows the inclusion of several similar plans in one tree-structured plan. The tree-structured plan consists of a single antecedent and several consequents organized in tree structures, as shown in Figure 8. To

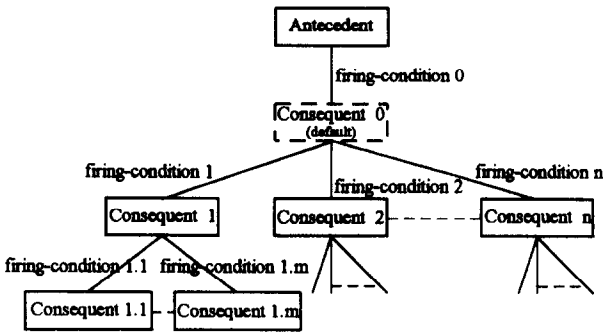

Figure 8. The tree structure of a plan.

select a specific tree-structured 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 a suitable consequent. The more general a consequent, the closer it is to the root of its tree. 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 an event matches an antecedent and **firing-condition 0** of the tree-structured plan is satisfied, the search for an appropriate consequent starts at the appropriate root, going down in the tree as far as possible. The path between a parent and a child can only be taken if the **firing-condition** associated with the child consequent is satisfied.

3.1.4 The basic analysis strategy. We have designed analysis techniques to provide mechanical assistance for the generation of formal specifications of different loop classes in many domains. We have applied these analysis techniques to some loops in the domain of scheduling university courses and the domain of basic algorithmic structures (Abd-El-Hafiz, 1994). It should be emphasized that our goal is to have analysis techniques that are automatable and flexible enough to be tailored to the needs of many domains. It is not our goal, even if it were possible, to handle all the cases that can occur in all possible domains. We have demonstrated the feasibility of automating our knowledge-based analysis approach by designing a prototype tool that annotates loops with predicate logic annotations (Abd-El-Hafiz, 1994; Abd-El-Hafiz and Basili, 1994).

In this article, we only describe the analysis of flat, simple *while* loops with noncomposite conditions. This description conveys the basic ideas behind our analysis strategy and demonstrates how to automatically generate formal specifications to assist in the understanding of code components in a specific do-

main of interest. The analysis of the other loop classes is described elsewhere (Abd-El-Hafiz, 1994).

Flat, simple loops with noncomposite conditions are annotated with their functional abstractions in a step-by-step process consisting of several phases, as depicted in Figure 9.

The first analysis phase, which is the symbolic execution of the loop body, abstracts a program's language- and implementation-specific features. For a detailed description of how to perform this symbolic execution, the reader is referred elsewhere (Abd-El-Hafiz, 1990; Mills et al., 1987; Zelkowitz, 1990). Using the notation introduced by Harandi and Ning (1990) about the possible abstraction levels of a program, this phase maps the source code to an implementation abstraction of the loop.

The second phase performs data flow analysis to produce an explicit representation of the dependencies among loop components in the form of BEs and AEs. Although this decomposition is different from the loop decomposition method introduced by Waters (1979), it was inspired by his work. Using the aforementioned notation introduced by Harandi and Ning (1990), this analysis phase reveals the structure of a loop and maps its implementation abstraction to a structural abstraction.

Finally, an analysis of the BEs and AEs, using a plan knowledge base, transforms the structural abstraction of a loop into a functional abstraction. The functional abstraction reveals the logical, as opposed to the syntactical or structural, details of the loop using predicate logic assertions.

The details of the analysis method are explained below. The descriptions of the analysis steps are interspersed with their application on the loop shown in Figure 10, which is the inner loop of the bubble sort algorithm. This loop is simple because the variable j is a unique control variable, and its modification is independent of the values of any other variables modified within the loop.

The first analysis phase symbolically executes the

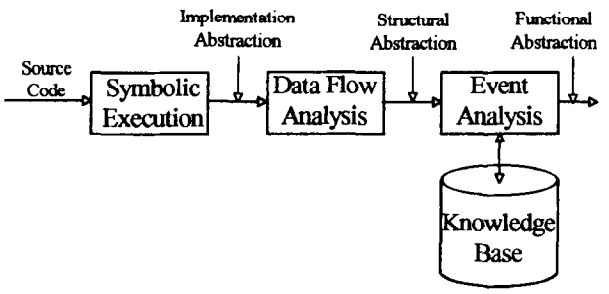

Figure 9. Analysis of flat, simple loop structures.

```
j := 1;
while j <= k do begin
   if capacity[j] > capacity[j+1] then begin
      temp_capacity := capacity[j+1];
      capacity[j+1] := capacity[j];
      capacity[j] := temp_capacity
   end;
   j := j + 1
end;
```

Figure 10. Example of a flat, simple loop with noncomposite condition.

body of the loop. This symbolic execution summarizes the effect of the body on each of the variables assigned in the loop. As a result, the net modification performed on each variable, in one loop iteration, is given in the form of a conditional assignment. For instance, assume that the loop body has n execution paths and $a_1, a_2, \ldots, a_n$ are the concurrent assignments that modify the variable v on each path. If $p_1, p_2, \ldots, p_n$ are the corresponding mutually exclusive predicates that determine which path should be taken, then the net modification performed on the variable v is given in the form $(p_1 \Rightarrow a_1), (p_2 \Rightarrow a_2), \ldots, (p_n \Rightarrow a_n)$. If some a_i does not modify v, then we remove the corresponding guarded concurrent assignment $(p_i \Rightarrow a_i)$ to simplify the conditional assignment.

Applying this analysis phase on the loop of Figure 10 yields the following set of conditional assignments. Each conditional assignments independently encapsulates the effect of the body on a unique variable. These statements represent an implementation abstraction of the loop body, which eliminates a program's language- and implementation-specific features.

$$true \Rightarrow j := j + 1$$

$$capacity[j] > capacity[j+1] \Rightarrow temp\_capacity := capacity[j+1]$$

$$capacity[j] > capacity[j+1] \Rightarrow capacity[j+1],\ capacity[j] := capacity[j],\ capacity[j+1]$$

The second phase decomposes the loop using data flow analysis to produce an explicit representation of the dependencies among loop components in the form of BEs and AEs. That is why the output of this phase is called a structural abstraction of the loop.

The BE of the loop is constructed by taking the loop condition B as its condition. If the variable *var* is the control variable, then the enumeration is the part of the symbolic execution outcome that modifies *var*. The initialization, if any, provides the initial value of the control variable before the start of the loop. The BE of the loop is recognized first because

the analysis of the remaining loop events depends on the result of its analysis.

The symbolic execution result is decomposed into AEs by first removing the enumeration part of the BE. Then, the minimal sets of conditional assignments, which are interdependent with respect to data flow and do not have data flowing out of them into other parts of the loop body, are recursively identified and isolated (Waters, 1979). The initialization, if any, provides the initial values of the variables modified in the augmentation body. The resulting AEs are ordered, after the BE, such that the ones identified first are analyzed last.

The application of the second analysis phase to our example yields the three ordered events given below. Because the variable j is responsible for the data flow out of the BE conditional assignment and into the AE's conditional assignment, the BE is ordered first. Similarly, because *capacity* is responsible for the data flow out of the second AE conditional assignment and into the third event conditional assignment, they have this specific order. This ordering makes it possible to propagate the effect of analyzing an event to the analysis of other events dependent on it.

1. BE
   condition: $j \leq k$
   enumeration: $j := j + 1$
   Initialization: $j := 1$
2. AE 1
   body: $capacity[j] > capacity[j + 1] \Rightarrow capacity[j + 1], capacity[j] := capacity[j], capacity[j + 1]$
   initialization:—
3. AE 2
   body: $capacity[j] > capacity[j + 1] \Rightarrow temp\_capacity := capacity[j + 1]$
   initialization:—

Finally, we try to match the loop events with the antecedents of the plans stored in the knowledge base. The matching results are the name of the unique plan matched along with the unification of the # terms in the plan with the actual values in the event. To represent these event-matching results, we use the *analysis knowledge* notation. The analysis knowledge (AK) of a variable modified by a certain event consists of an n-tuple where n is dependent on the specific plan matched. The first term of the tuple is the name of the plan matched. The remaining $(n - 1)$ terms are the results of matching the # variables with the actual values in the event.

The instantiation of the consequents of the matched plans with the actual values gives the contribution of each individual event to the assertions of the loop. The parts of the loop that cannot be matched with a plan antecedent are printed to the user as parts that could not be handled. The knowledge base manager can consider adding plans to help in translating these events later.

Applying the third analysis phase to the results of the data flow analysis matches the first two events with the antecedents of the plans shown in Figures 6 and 7, respectively. The third event is matched with a plan that is not shown here due to space limitations. This plan discards the temporary variable, *temp_capacity*, information by having *true* predicates as annotations. The analysis knowledge of the variables j and *capacity* are as follows:

1. $AK(j) = (DBP_1, var\#: j, var_0\#: 1, R\#: \leq, exp\#: k)$
2. $AK(capacity) = (SLAP_1, var\#: j, array\#: capacity, exp\#: j + 1, FUNC\#: PRED)$

Instantiating the consequents of the identified plans with the actual values yield the following results:

1. **precondition** $0 \leq k$
   invariant $1 \leq j \leq k + 1$
   postcondition $j = k + 1$
   sequence $1..j - 1$
   final-sequence $1..k$
2. **precondition** *true*
   invariant $perm(capacity, capacity_0) \wedge maximum\_at\_edge(capacity[1..j - 1], j)$
   postcondition $perm(capacity, capacity_0) \wedge maximum\_at\_edge(capacity[1..k], k + 1)$

The functional abstraction of the loop is synthesized from the instantiated plans' consequents. The precondition, invariant, and postcondition are constructed by taking the conjunction of the corresponding parts in the instantiated consequents. For instance, the functional abstraction of the loop in Figure 10 is given below. These are the specifications of the inner loop of Figure 1 if it is analyzed in isolation of the outer loop surrounding it. When we analyze the whole nested construct, some predicates might be added to the inner loop specifications, as explained in the next subsection.

Precondition: $0 \leq k$
Invariant: $1 \leq j \leq k + 1 \wedge perm(capacity, capacity_0) \wedge maximum\_at\_edge(capacity[1..j - 1], j)$

Postcondition: $j = k + 1 \wedge perm(capacity, capacity_0) \wedge maximum\_at\_edge(capacity[1..k], k + 1)$

One advantage of the resulting specifications is that they explain the function of the algorithm in concise terms without any ambiguities. They explain what the algorithm does without interleaving it with information on how it does it. To further investigate the details of how a specific loop is designed, the decomposition of the loop into events and the augmentation of the individual events with their individual specifications can be used. In other words, this documentation technique supports a top-down strategy of program understanding (Brooks, 1983). In this strategy, a general understanding of a loop, which is formed using the formal specifications, can be refined and elaborated based on information extracted from the loop decomposition and analysis results of the individual events.

The readability of the resulting specifications is improved without affecting their semantic soundness. This is because each new term still has an underlying rigorous definition that can be used if desired. To further improve the readability of these abstract specifications in a specific domain of interest, some of the commonly occurring predicates can be replaced with domain-specific ones. For example, in the domain of scheduling university courses, these specifications can be written in terms of the predicate *largest_room_at_edge* instead of *maximum_at_edge*.

The domain-specific replacements can be done explicitly 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, which 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.

3.1.5 Beyond the basic analysis strategy. The analysis of general loops and loops with composite conditions is performed using steps similar to those described in the previous section (Abd-El-Hafiz, 1994; Abd-El-Hafiz and Basili, 1993). The formation of the events and the synthesis of the event analysis results take into account the fact that there might be more than one control variable and/or BE. In general loops, we can produce loop preconditions and invariants that assist in the understanding and verification activities. Because the control computation of general loops is not as determinate and isolated as in the case of simple loops, we do not produce some of the specific analysis results that were produced for simple loops. The sequences of values assumed by the control variable(s) and the program state at the end of the loop are usually dependent on combined indeterminate effects of several events and the values of some program variables (Abd-El-Hafiz and Basili, 1993). As a result, the plans that analyze general loops neither include the aforementioned sequences nor use them in writing loop specifications. The postcondition can only be deduced after the synthesis of the loop invariant. The postcondition is formed by taking the conjunction of the loop invariant and the negation of the loop condition (Hoare, 1969). Using this method to obtain the loop postcondition yields predicates that might not be as informative and concise as those of simple loops. This, in turn, makes the resulting postcondition less easy to read and understand. Here, the simplification of the resulting predicates and the use of domain-specific abstractions are even more important.

Nested loops are analyzed by recursively analyzing the innermost loops and replacing them with sequential constructs that represent their functional abstraction (Abd-El-Hafiz, 1994). The resulting specifications of the outermost loop, as well as those for the inner ones, are used to understand the whole nested construct. However, if we are interested in more than understanding and documentation and want to enable the proof of Hoare verification conditions (Hoare, 1969),then the inner loop specifications might need some modifications. Because our recursive analysis approach is performed bottom-up, and complete knowledge of the inner loop functions is available during the analysis of outer loops, the generated outermost loop specifications enable the proof of Hoare verification conditions. On the other hand, inner loops are analyzed in isolation of the outer ones enclosing them; consequently, their invariants might not be strong enough to satisfy some Hoare verification conditions. An additional adaptation phase is, hence, designed to strengthen inner loops invariants by adding some context-related predicates to them. For instance, when the inner loop of the bubble sort example is analyzed in isolation, as in the previous subsection, its invariant does

not include any information about the sorted segment of the array *capacity*. Thus, the adaptation phase strengthens the inner loop invariant by adding the predicate $\forall k + 2 \leq ind \leq num\_of\_rooms$: $capacity[ind] \geq capacity[1 .. ind - 1]$, which can be abstracted to $upsorted(capacity[1 .. num\_of\_rooms], k + 2)$ (Figures 3 and 4). By providing information about the sorting context of the inner loop, this predicate enables the verification of the whole nested construct. However, the addition of the correct predicates to the inner loop invariants is not always possible in the case of nested constructs in which an inner loop is preceded by statements other than assignment and conditional statements (e.g., loops or procedure calls). Thus, this theoretical limit affects the ability to prove the resulting specifications. For more details on the analysis of nested loops and the limitations of our approach, see Abd-El-Hafiz (1994).

3.1.6 Evolution of the knowledge base. The success of the developed analysis techniques in a specific application domain is dependent, to a great extent, on the design of efficient and correct plans. That is why the tasks of designing plans and managing the knowledge base for a specific domain of interest should be performed by someone expert in both the desired domain and formal specifications.

To create a knowledge base, the desired domain should be analyzed to design an initial set of plans believed to cover a considerable number of loop constructs that might occur in it. After adding this initial set, the knowledge base should evolve over time. It should undergo a process of controlled usage in which the knowledge base manager needs to closely monitor its use.

The basic understanding of a domain is represented in the initial set of plans constituting the knowledge base. Further use of the knowledge base is apt to reveal inadequacies in it with respect to the sufficient number of plans, their structure, and their abstraction level. This use is also likely to improve the understanding of the domain and increase the knowledge of its details. That is why a controlled use of the knowledge base is needed to adapt the plans and make their abstraction level, structure, number, and naming conventions suitable for the domain under consideration. For example, a failure to identify the specification of a loop event indicates that either the event is erroneously designed and requires modification, or that there is a missing plan, which should be added to the knowledge base. The user needs to check the unspecified event to see if he or she can modify it. If no error is detected, then the knowledge base manager is notified. Whenever a new plan needs to be added to the knowledge base, it should be investigated whether to add it as an independent plan or to improve on the structure and/or the knowledge represented in the existing plan(s) to cover the new case.

We performed a case study manually on a complete set of loops in a real program of some practical value. Thus, case study results are not affected by an implementation limits. This case study served to test our analysis techniques and evaluate their strengths and weaknesses within a specific application domain. Given this fixed set of loops, the small number of plans needed to analyze them demonstrated the positive effect our analysis techniques could have on the size of the knowledge base.

The program chosen for the case study is in the domain of scheduling university courses (Jalote, 1991). It has 1,400 executable lines of code and 77 loops. This program deals with scheduling a set of courses. During this case study, we had to analyze and specify loops that use data types such as pointers and that have a variety of PASCAL statements.

We gradually populated the knowledge base with plans. First, we decomposed every loop under consideration into the BEs and AEs. Then, we analyzed every event 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, then we designed a new plan with initial specifications. We then modified the plan and tailored it to give correct specifications by trying to prove the loop invariant using Hoare techniques (Hoare, 1969). If a plan that matched a similar event, but not the exact one under consideration, existed in the knowledge base, then we considered further improvements on the structure and/or knowledge represented in the existing plan.

Out of the 77 loops, we completely analyzed 65 (84.4%) and partially analyzed 12. We only designed 48 plans to analyze the 213 events of the completely analyzed loops and 22 events from the partially analyzed ones. We decided not to specifically design plans for the analysis of 12 loops in the case study. The unique and complex nature of these loops suggested that the effort needed to design plans for their analysis highly outweighs the advantages that could be gained by using the plans in this specific application domain. These 12 loops were analyzed using the available set of plans to determine whether useful partial specifications could be obtained. If these loops were common in another domain, it

Table 2. Numbers and Percentages of Completely Analyzed Loops Along the Three Dimensions

	Dimension					
	1		2		3	
Analysis statistics	Simple Loop	General Loop	Noncomposite Condition	Composite Condition	Flat Body	Nested Body
Available number	52.0	25	46.0	31.0	53.0	24.0
Number analyzed	48.0	17	42.0	23.0	52.0	13.0
Percentage analyzed	92.3	68	91.3	74.2	98.1	54.2

would have been worthwhile to invest some effort in designing their plans.

Table 2 gives the number of completely analyzed loops in each class defined by our taxonomy. Along each dimension, the variation between the percentages of completely analyzed loops in the two complementary classes is somewhat large. A possible interpretation for these variations is that they indicate what classes are more appropriately (or easily) analyzed by our analysis techniques. The variations along the three dimensions indicate that simple (flat) loops are considerably easier to analyze than general (nested) loops. They also indicate that composite loop conditions make the analysis harder than noncomposite conditions.

Table 3 shows the number of plans designed and the number of events they analyze. Table 4 shows the same information for the abstracted/tree-structured plans. The average and standard deviation of the number of utilizations of the 48 plans are 4.9 and 7.97, respectively. The average and standard deviation of the number of utilizations of the 10 abstracted/tree-structured plans are 14.9 and 11.8, respectively. More specifically, the 10 abstracted/tree-structured plans (20.8%) analyzed 149 events (63.4%) out of the 235 events analyzed in this study. These plans have a total of 24 consequents and underwent 8 abstractions. This means that the experience gained during this case study enabled us to encapsulate the knowledge of at least 32 (24 + 8) simple plans into 10 deep and well-developed ones. This encapsulation, in turn, led to a considerable reduction in the size of the knowledge base. Coming up with such a set of abstracted/tree-structured 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 more concise and useful plans.

To examine the limits of our loop analysis approach, we investigated the characteristics of the 12 partially analyzed loops. Almost all of them (11 out of 12) are nested and contain procedure and function calls (10 out of 12). The average number of procedure and function calls per loop is 4.6 (standard deviation, 2.6). In Table 5, variations between the characteristics of the partially analyzed loops and the completely analyzed ones are highlighted. For more details on the differences between the 12 partially analyzed loops and the 65 completely analyzed ones, see the source code listings in Abd-El-Hafiz (1994) and Jalote (1991).

The 12 partially analyzed loops contain relatively high numbers of events, lines of code, modified variables, and procedure and function calls. These factors increase the difficulty of designing the invariants and, consequently, the plans. For instance, to design plans for the analysis and specification of a loop containing procedure and function calls, all the procedures and functions called must first be formally analyzed using Hoare techniques (Hoare,

Table 3. Utilization of the Designed Plans

	Plan Category	
Analysis Statistics	BP	AP
Number of plans	11	37
Number of utilizations	95	140

Table 4. Utilization of the Abstracted and/or Tree-Structured Plans

	Plan Category	
Analysis Statistics	BP	AP
Number of plans	4	6
Number of utilizations	75	74

Table 5. Comparison Between the Completely and Partially Analyzed Loops

Characteristics*	Completely Analyzed Loops	Partially Analyzed Loops
Events	3.28 (SD = 2.05)	11.92 (SD = 4.77)
Executable SLOC	10.45 (SD = 8.29)	43.2 (SD = 15.7)
Modified variables	3.42 (SD = 2.45)	12.4 (SD = 4.9)

* In terms of average numbers

1971). The theoretical limit discussed in the previous section, which is related to the adaptation of inner loop specifications in nested loops for the purpose of proving their correctness, occurred in only one loop. That is, the partial analysis of the remaining 11 loops in this case study is attributed to practical limits. These practical limits stem from the plan designer's inability to formally analyze complicated loops and find their invariants despite the fact that these invariants exist theoretically.

4. CONCLUSION

We have discussed the documentation language characteristics that are required to facilitate the understanding of a code component. To mechanically generate program documentation that has these characteristics, we have presented a hybrid analysis approach that combines the use of knowledge-based and algorithmic analysis approaches. The knowledge-based analysis approach produces formal loop specifications that can be used by the algorithmic approach to produce formal specifications of complete program modules. Our hybrid analysis approach generates formal specifications that have the required semantic soundness and expressive power. This is because they have a sound mathematical basis and can abstract a wide variety of problems. The readability of the resulting specifications is improved by abstracting them using high-level and/or domain-specific terms. Furthermore, the systematic techniques that we developed for producing these specifications improve their automatability (Abd-El-Hafiz, 1990, 1994).

To assist in the algorithmic analysis of complete programs, we focused on explaining how to analyze loops to produce documentations that are more formal and accurate than those produced by other approaches (Harandi and Ning, 1990; Hartman, 1991; Rich and Wills, 1990). Our approach analyzes stereotyped loop fragments that have nonadjacent parts and, consequently, avoids the large size of the knowledge base needed to compensate for this drawback (Letovsky, 1987). It presents well-defined methods for selecting the fragments to be analyzed and for choosing the rules that analyze them. Hence, it relieves the user from the difficulty of having to perform this task on a code that is not well understood (Ward et al., 1989).

Our analysis approach supports software development by providing code abstractions that can help in both reviewing and debugging code. In many cases, it can also help in proving the correctness of loop implementations. By assisting in the rigorous understanding of loops, our approach facilitates code maintenance. Although our approach is useful for analyzing and understanding most loops, our case study showed that the analysis of some loops may be beyond the endurance of the analyzer. However, a maintainer who needs to understand a particular code segment may be more willing to pursue the analysis further or settle for less formal understanding. With respect to software reuse, our approach facilitates the population of a software repository with well-documented code components.

By performing a case study, we were able to study the effect of the analysis techniques on the size of the knowledge base and to package our experience in the design and use of plans for a specific domain. We have developed a prototype tool that implements the knowledge-based loop analysis approach (Abd-El-Hafiz and Basili, 1994). We had earlier reported a prototype tool that uses user-supplied loop annotations for analyzing complete programs (Abd-El-Hafiz et al., 1991). The integration of these two tools to develop a larger system that performs intelligent analysis of complete program modules needs to be investigated. The practicality of our approach should be further investigated by testing them in various domains and developing domain-specific abstractions.

ACKNOWLEDGMENTS

We thank Gianluigi Caldiera, Sandro Morasca, and Dieter Rombach for their helpful contributions to a variety of aspects of this article.

This research was supported in part by Office of Naval Research grant NOOO14-87-k-0307 to the University of Maryland.

REFERENCES

Abd-El-Hafiz, S. K., A Tool for Understanding Programs Using Functional Specification Abstraction, Master's Thesis, University of Maryland, College Park, Maryland, 1990.

Abd-El-Hafiz, S. K., A Knowledge-Based Approach to Program Understanding, Ph.D. Thesis, University of Maryland, College Park, Maryland, 1994.

Abd-El-Hafiz, S. K., and Basili, V. R., Documenting programs using a library of tree structured plans, in *Proceedings of the Conference of Software Maintenance*, 1993, pp. 152–161.

Abd-El-Hafiz, S. K., and Basili, V. R., A tool for assisting the understanding and formal development of software, in *Proceedings of the Sixth International Conference on Software Engineering and Knowledge Engineering*, 1994, pp. 36–45.

Abd-El-Hafiz, S. K., Basili, V. R., and Caldiera, G., Towards automated support for extraction of reusable

components, in *Proceedings of the Conference of Software Maintenance*, 1991, pp. 212–219.

Basili, V. R., and Mills, H. D., Understanding and Documenting Programs, *IEEE Trans. Software Eng.* SE-8, 270–283 (1982).

Basu, S. K., and Misra, J., Proving Loop Programs, *IEEE Trans. Software Eng.* SE-1, 76–86 (1975).

Bertels, K., Vanneste, P., and De Backer, C., A cognitive approach to program understanding, in *Proceedings of the Working Conference on Reverse Engineering*, 1993, pp. 1–7.

Brassard, G., and Bratley, P., *Algorithmics: Theory & Practice*, Prentice-Hall, 1988.

Brooks, R., Towards a Theory of the Comprehension of Computer Programs, *Int. J. Man-Machine Stud.* 18, 543–554 (1983).

Dunlop, D. D., and Basili, V. R., A Comparative Analysis of Functional Correctness, *Comp. Surv.* 14, 299–244 (1982).

Dunlop, D. D., and Basili, V. R., A Heuristic for Deriving Loop Functions, *IEEE Trans. Software Eng.* SE-10, 275–285 (1984).

France, R. B., and Basili, V. R., A Pattern-Driven Approach to Code Analysis for Reuse, Technical Report CS-TR-2802, Department of Computer Science, University of Maryland, College Park, Maryland, 1991.

Good, D., Mechanical proofs about computer programs, in *Mathematical Logic and Programming Languages* (C. A. R. Hoare and J. C. Shepherdson, eds.), Prentice-Hall International, 1985, pp. 55–75.

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

Harandi, M. T., and Ning, J. Q., Knowledge-Based Program Analysis, *IEEE Software* 7, 74–81 (1990).

Hartman, J., Understanding natural programs using proper decomposition, in *Proceedings of the 13th International Conference on Software Engineering*, 1991, pp. 62–73.

Hausler, P. A., Pleszkoch, M. G., Linger, R. C., and Hevner, A. R., Using Function Abstraction to Understanding Program Behavior, *IEEE Software*, 7, 55–63 (1990).

Hayes-Roth, F., Rule-Based Systems, *Commun. ACM* 28, 921–932 (1985).

Hoare, C. A. R., An Axiomatic Basis for Computer Programming, *Commun. ACM* 12, 576–580, 583 (1969).

Hoare, C. A. R., Procedures and parameters: An axiomatic approach, in *Symposium on the Semantics of Algorithmic Languages*, 1971, pp. 102–116.

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

Jensen, K., and Wirth, N., *Pascal User Manual and Report*, Springer-Verlag, 1985.

Johnson, W. L., and Soloway, E., PROUST: Knowledge-Based Program Understanding, *IEEE Trans. Software Eng.* SE-11, 267–275 (1985).

Katz, S., and Manna, Z., Logical Analysis of Programs, *Commun. ACM* 19, 188–205 (1976).

Kemmerer, R. A., and Eckmann, S. T., UNISEX: A UNIx-based Symbolic EXecutor for Pascal, *Software Pract. Exp.* 15, 439–458, (1985).

Letovsky, S., Program understanding with the lambda calculus, in *Proceedings of the 10th International Joint Conference on AI*, 1987, pp. 512–514.

Mills, H. D., The New Math of Computer Programming, *Commun. ACM* 18, 43–48 (1975).

Mills, H. D., Basili, V. R., Gannon, J. D., and Hamlet, R. G., *Principles of Computer Programming: A Mathematical Approach*, Allyn and Bacon, 1987.

Misra, J., Some Aspects of the Verification of Loop Computations, *IEEE Trans. Software Eng.* SE-4, 478–486 (1978).

Morris, J. H., Jr., and Wegbreit, B., Subgoal Induction, *Commun. ACM* 20, 209–222 (1977).

Qian, S. S., A Tool for Understanding Software Components, Master's Thesis, University of Maryland, College Park, Maryland, 1989.

Quilici, A., A hybrid approach to recognizing programming plans, in *Proceedings of the Working Conference on Reverse Engineering*, 1993, pp. 126–133.

Remmers, J. H., A Technique for Developing Loop Invariants, *Inf. Proc. Lett.* 18, 137–139 (1984).

Rich, C., and Waters, R. C., Formalizing reusable software components in the programmer's apprentice, in *Software Reusability*, vol. II (T. J. Biggerstaff and A. J. Perlis, eds.), ACM Press, 1989.

Rich, C., and Wills, L. M., Recognizing a Program's Design: A Graph-Parsing Approach, *IEEE Software* 7, 82–89 (1990).

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

Soloway, E., and Ehrlich, K., Empirical Studies of Programming Knowledge, *IEEE Trans. Software Eng.* SE-10 (1984).

Soloway, E., Bonar, J., and Ehrlich, K., Cognitive Strategies and Looping Constructs: An Empirical Study, *Commun. ACM* 26, 853–860 (1983).

Ward, M., Calliss, F. W., and Munro, M., The maintainer's assistant, in *Proceedings of the Conference on Software Maintenance*, 1989, pp. 307–315.

Waters, R. C., A Method for Analyzing Loop Programs, *IEEE Trans. Software Eng.* SE-5, 237–247 (1979).

Wegbreit, B., The Synthesis of Loop Predicate, *Commun. ACM* 17, 102–112 (1974).

Zelkowitz, M. V., The Functional Correctness Model of Program Verification, *IEEE Comp.* 30–39 (November 1990).