Packaging Reusable Components: The Specification of Programs*

V. R. Basili S. K. Abd-El-Hafiz

Department of Computer Science,
University of Maryland,
College Park, MD 20742, U.S.A.

September 10, 1992

Abstract — Packaging software components, before storing them in a reuse repository, is an important activity which could significantly enhance their reuse potential. The packaging of code components can be achieved by annotating them with their abstract formal specifications. Such specifications can facilitate their identification, understanding, and modification during the actual reuse process. This paper presents a knowledge-based approach that helps in annotating computer programs by finding the abstract specifications of their loops. In this approach, loops are first decomposed into their component parts. The specifications of the resulting loop fragments are obtained by utilizing a knowledge-base. This knowledge-base is built, in a specific domain, by designing plans that allow us to recognize stereotypical code patterns and to associate them with their specifications. A consistent and accurate specification of the whole loop construct is then synthesized from the understanding of its fragments. An example is provided to illustrate this technique for a program with a simple loop structure.

*Research for this study was supported in part by the ONR grant N00014-90-J-4091 to the University of Maryland.

Contents

1 Introduction

The reuse of all forms of proven experience can improve software quality and productivity considerably[5]. With respect to product reuse, properly packaging software products in a reuse repository can enhance their reusability. The objective of finding suitable computer-based techniques to package reuse candidates is to increase their reuse potential by facilitating their identification, understanding, and modification during the actual reuse process. The different packing activities which can be performed on a component can range from generalizing the component's interface by eliminating its dependency on other software to representing it in a different and more abstract form. In any case, the augmentation of the component's interface with the specifications necessary for its correct understanding is crucial for reuse. This is because the reuser must be aware of what the component does and how to utilize it. These specifications can also assist in identification and modification.

In this paper, we discuss the different packaging activities performed on candidate reusable components and perform a comparison of some of the possible formal and informal languages available for specifying them. We argue that it is possible to produce readable abstract specifications which have an underlying sound mathematical foundation and present a knowledge-based approach for producing such specifications. In this approach, loops are classified according to their complexity levels. Based on this taxonomy, we design the analysis techniques that best fit each of these classes. Loops are analyzed by decomposing them into their component parts. This decomposition is based on the structural dependencies among the different loop fragments. To deduce the abstractions of these loop fragments, a knowledge-base of plans is utilized. After deducing the abstractions of the loop fragments, a consistent and accurate abstraction of the whole loop construct is synthesized from the understanding of its constituents.

2 The Reuse Framework

Reuse is motivated by the desire to improve quality and productivity. Improving quality can be achieved by reusing all forms of proven experience related to software development including products, processes, and other knowledge. Improving productivity can be achieved by using all forms of existing experience rather than creating everything from scratch[6].

Here, the term 'product' refers to a concrete documentation or artifact created during a software project and the term 'process' refers to a concrete activity or action performed by a human being or a machine during the software development. The reuse of products includes such activities as reusing code components, requirements, and design templates developed for some application domain. The reuse of software processes include such activities as adapting a development method and reusing an inspection method created for similar prior projects. The reuse of other knowledge refers to utilizing anything useful for software development including models, data, and learned lessons. It includes such activities as developing a risk management plan based upon lessons learned from applying a new technology and estimating the cost of a project based on data collected from past projects.

With respect to product reuse, the reuse of early life cycle products such as specifi-

cations and requirements might provide the largest payoff since errors in the early stages of development are more costly than other kinds of errors. To avoid such errors, the Requirements Apprentice (RA) system[33] assists a requirements analyst in the creation and modification of software requirements by utilizing a reuse library of requirements clichés. When creating a requirement's document, the information unique to the particular problem comes from the analyst while the bulk of general information about the domain comes from the cliché library. Another form of product reuse is the reuse of software designs. In the Intelligent Design Aid (IDeA) system[26], software design components are abstracted into the form of design schemas to enhance their reusability. IDeA provides an environment in which the user can concentrate on supplying the requirements and specifications of the desired software product and is assisted in the location and incorporation of the reusable components to construct the required software design. At the lowest level, highly controlled source code repositories can provide quick gain. The code components designed by Booch[9] are an example of what could be stored in a general purpose repository. They include implementations of abstract data types as well as numerous algorithms for sorting, searching, and pattern matching which can appear in a vast spectrum of applications.

Whether the components in the repository are created from scratch or extracted and adapted from existing programs and systems, three main activities need to be performed to reuse them. These activities are the identification of the reuse candidates from the repository, the understanding and selection of the best suited candidate, and the modification of the selected candidate into the required one.

To identify reuse candidates, several approaches have been proposed. The faceted classification scheme[30] describes, and selects, software components by their functionality, internal environment (where the program is executed), and external environment (where the program is applied). Other approaches, as in the Paris system[22], adopt logic-based identification of components. In this system, the identification utilizes preconditions and postconditions plus assertions about component properties to form clauses to be proved with the Boyer-Moore theorem prover. The result of this formal process gives a list of candidate components.

The understanding of a component is required whether or not it is to be modified because the users need a mental model of the component's computation to use it properly[8]. The annotation of components with their specifications can help the users in forming this mental model, thus selecting the components best suited for their needs[1, 2, 13].

To allow for significant reuse, the modification of selected components prior to their inclusion in a new system must be anticipated. This modification can take place as part of the actual reuse process or as part of the packaging activity of the components before storing them in the repository[6].

To enhance the reusability of candidate reusable components, they must be packaged in a form which facilitates the performance of the aforementioned three activities. That is why packaging of candidate reusable components is one of the important issues that needs to be addressed in software reuse. Finding suitable computer-based techniques to package reuse candidates can facilitate their understanding and selection. It can assist in their modification during the actual reuse process and can help in the identification of candidate

reusable components from a repository.

3 The Different Packaging Activities

A rich and well-organized catalog of high-quality reusable components is the key to a successful component repository and a long term economic gain. But developing reusable components is generally more expensive than developing specialized code because of the overhead of designing for reusability and maintaining the component repository. However, such a catalog can be available to an organization if it can reuse the same code it developed in the past. Mature application domains, where most of the functions that need to be used already exist in some form in earlier systems, should provide enough components for code reuse. For example, Lanergan and Grasso found rates of reuse of about 60% in business applications[24].

After extracting a component from an existing system, it needs to be properly packaged before adding it to the repository. The packaging of candidate reusable objects can enhance their reusability. This is because it can facilitate their identification, their understanding, and their modification during the actual reuse process.

Interface generalization is necessary for producing independent objects. This generalization can be performed by completing an object so that it has all the external references needed to reuse it independently[10]. The use of the the generic feature of Ada is another common technique for producing more independent and, presumably, more reusable components[3]. However, The reusability of an object can be further enhanced by permitting the interface to include more than purely syntactic information. The interface of reusable objects can include information which is needed for their correct understanding. It can be designed to facilitate their modification, composition and integration into new systems[14].

The CARE system(Computer Aided Reuse Engineering)[10] has been designed to support a process model for extracting resuable software components from existing systems and then adapting them. It populates the reuse library using software metrics to produce a first assessment of independence, quality, and reuse potential to generate a set of candidate components. Then, a human expert processes the candidates and packages a subset based on their functionality and quality. To start this second phase, a precise description of what the component does is generated. While doing this, the correctness of the implementation is also verified. Test cases are then generated, executed, and associated with the component. The formal specification previously produced can help in this activity. Finally, the extracted candidates are stored in the component repository along with their functional specification and test cases and manuals are provided for them.

Choosing an appropriate formalism for representing software components is another factor that can improve reusability. An effective representation of software components can offer several advantages. It can enhance the modifiability of components by introducing convenient instantiation and combination operations[20]. It can enhance program reliability by defining a set of constraints with each component that can be used in maintaining its internal consistency. Furthermore, an abstract formalism makes it possible to repre-

sent components in such a way that they can be easily reused in many different language environments.

One formalism for representing code components is the Plan Calculus[34]. The Plan Calculus is essentially a combination of hierarchical data flow schemas, logical formalisms, and transformations. Some of the useful properties of such a formalism are: programming language independence, semantic soundness, and convenient combinability. KBEmacs (Knowledge-Based Editor in Emacs)[34] is a program editor that makes it possible to construct programs rapidly and reliably by combining algorithmic components called *clichés*. It is based on a simple, early plan representation that corresponds to a part of the Plan Calculus. KBEmacs supports several commands for instantiating and combining components from the component library. The components themselves are stored in the library as plans. New components can be defined by the user by using a programming language-like representation.

While the approach of representing candidate reusable objects in an abstract form[34] offers some advantages over other techniques, which only generalize the candidate reuse objects by eliminating their dependency on other software[3], they are much more expensive. The two approaches need to be evaluated in the environment in which they are going to be used in order to determine which one best fits the needs of the organization. However, in both cases, the augmentation of the object's interface with the necessary specifications needed for its sound and correct understanding[1, 2, 13] is crucial for reuse. This is because the reuser must be aware of what the component does and how to utilize and modify it. Furthermore, the specification of a candidate reusable component can help in its identification as demonstrated by the Paris system[22]. To choose the appropriate language that can be used in specifying reusable components, we need to perform a comparison of the various choices of specification languages. The next section discusses the various specification mechanisms and compares the possible choices for a specification language.

4 The Specification of Programs

Due to the importance of the activity of program comprehension, substantial interest is directed towards the analysis and specification of computer programs. The algorithmic approaches usually annotate programs according to the formal semantic of a specific model of correctness[1, 23]. A common characteristic of the systems that implement these approaches, up to date, is that they rely on the user in the task of annotating the loops. They offer assistance only in proving the correctness of these annotations.

The knowledge-based approaches modularize experts' knowledge in the form of plans that can be accessed mechanically. In these approaches, the generation of the specification of a component usually involves two main tasks; the recognition of stereotypical parts in the program and deriving their annotations using the plans stored in the knowledge-base. In the graph parsing-approach, a program is translated into a graph. Then it is parsed using a library of plans in the form of graphical grammer rules[35]. The heuristic-based concept-recognition approach provides common-sense explanations of the program that trades accuracy for simplicity[17]. The transformational approach of program analysis

is similar to the transformational paradigm of automatic program synthesis but with the application direction of the transformation rules reversed[25].

Given the wide range of specification languages available, the choice of a language which is suitable for specifying candidate reusable components is crucial. Specification languages can range from free-form natural languages to abstract formal specifications. To improve the reusability of a candidate component, the specification language should have many characteristics. Readability, expressiveness, semantic soundness, convenient derivability, and automatability are of particular importance.

- *Readability* is the convenience and ease with which a reuser can read the specifications and then form a sound understanding of the components functionality.
- *Expressiveness* is the ability of assisting the understanding process of as many different components as possible.
- *Semantic soundness* means that the specification language must be based on a mathematical foundation which allows correctness conditions to be stated and verified if desired.
- *Convenient derivability* is the ability to derive the specifications of various components using easy and well defined methods.
- *Automatability* is the ability to automatically generate and manipulate the specifications of various components.

A small, but representative, subset of the various specification languages that can augment a reusable component are discussed in this section. The relative strengths and weaknesses of the different possible choices are evaluated in light of the aforementioned characteristics. We demonstrate and and compare these different choices using the bubble sorting example shown in Fig. 1.

```
1    k, j, n, s: integer;
2    A: array[1..Max] of integer;

11   k := n - 1;
12   while k >= 1 do
13      j := 1;
14      while j <= k do
15         if A[j] > A[j+1] then
16            s := A[j+1];
17            A[j+1] := A[j];
18            A[j] := s;
19         fi
20         j := j + 1;
21      od
22      k := k - 1;
23   od
```

Fig. 1. A bubble sorting algorithm.

This is a bubble sorting 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 $A[1:n]$ is sorted in ascending order by repeatedly placing the maximum towards the end of the scanned segment.

Fig. 2. Free-form English specification of the bubble sorting algorithm.

Specifying programs using free-form English text is an informal technique that gives an intuitive description of the code. In most cases, the specifications are easy to read and the knowledge needed to produce and understand them is gained from programming courses and practical experience. The greatest strength of free-form English text is its expressiveness[34] since it is capable of specifying any kind of component. However, there is no semantic basis that makes it possible to determine whether or not the specifications have the desired meaning. This lack of a firm semantic basis makes informal natural language specifications inherently ambiguous. With respect to derivability, free-form English text might appear to be very easy to derive by just giving intuitive descriptions of the component under consideration. But since there is no well defined method for writing such specifications, the specifier must be very careful not to write an ambiguous statements that can lead to a misunderstanding. Due to their informal nature, judging the conciseness and clarity of natural language specifications is a difficult task that is dependent on the experience and talent of both the specifier and reuser. For instance, Fig. 2 shows a possible free-form English specification of the bubble sorting algorithm of Fig. 1. It gives a description of what is performed by the algorithm in addition to the undesired information on how it is performed. The parts of the array A 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, free-form English text is not amenable to automatic manipulation in any significant way[34]. In order to automatically generate informal specifications of programs, several knowledge-based approaches such as the graph parsing approach[35] and the heuristic-based concept-recognition approach[17] have been developed.

Other specification techniques produce natural language specifications that are more structured than the free-form ones[17]. These specifications give detailed descriptions of the purpose of each variable and statement in the program as outlined in Fig. 3. This form of informal specifications is similar to free-form English with respect to the five characteristics under consideration but they are more structured and detailed. They include unnecessary descriptions of how the sorting is performed. For example, the details of the swapping process, in item 2.2, are not needed for the sound and correct understanding of the loop. It seems that such detailed descriptions can encourage the reader to skip some parts of the specifications and consequently miss some important details.

The specifications shown in Fig. 4. are written in a formal notation that uses predicate calculus to produce Hoare-style annotations[18]. An advantage of formal specifications is that they accurately state what is performed by a program segment without having to give unnecessary details of how it is performed. Semantic soundness and expressiveness are key

The segment at lines 11-23 sorts an array A using a bubble sort algorithm.
The sorting is performed as follows:

1. The effect of lines 11, 12, 22, and 23 is to decrementally change the value of k from $n-1$ to 1.
2. The effect of lines 13-21 is to sequentially switch the adjacent elements in the array A if $A[j] > A[j+1]$, indexed by j from 1 to k.

This is performed as follows:

2.1 The effect of lines 13, 14, 20, and 21 is to incrementally enumerate the elements in the array A indexed by j from 1 to k.

It consists of:

2.1.1 $\cdots$

2.1.2 $\cdots$

2.2 The lines 15-19 switch the values in $A[j]$ and $A[j+1]$ if $A[j] > A[j+1]$. They consist of:

2.2.1 An IF construct at lines 15 and 19.

2.2.2 A swap construct at lines 16-18. It performs the following:

2.2.2.1 An assignment construct at line 16.

2.2.2.2 An assignment construct at line 17.

2.2.2.3 An assignment construct at line 18.

Fig. 3. Structured English specification of the bubble sorting algorithm.

Outer loop invariant:

$n-1 \geq k \geq 0 \ \wedge \ perm(A, A_0) \ \wedge$
$\forall\ k+2 \leq ind \leq n : A[ind] = maximum\{A[1 : ind-1]\}$

Inner loop invariant:

$1 \leq j \leq k+1 \ \wedge \ 1 \leq k \leq n-1 \ \wedge$
$A[j] = maximum\{A[1 : j-1]\} \ \wedge \ perm(A, A_0) \ \wedge$
$\forall\ k+2 \leq ind \leq n : \ A[ind] = maximum\{array[1 : ind-1]\}$

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

Fig. 4. Hoare-style specification of the bubble sorting algorithm.

advantages of the predicate calculus 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 makes it possible to be certain of the functionality of a given component.

However, 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[13]. An example of these abstractions is shown in Fig. 5 which abstracts

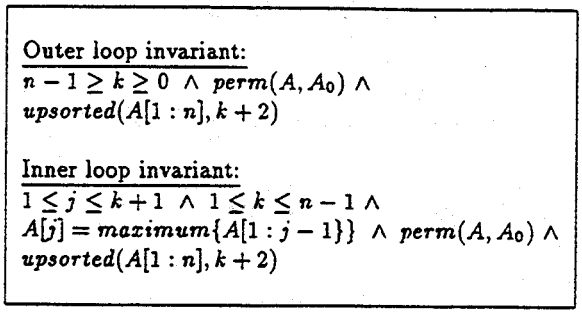

Fig. 5. Abstract specification of the bubble sorting algorithm.

the annotations in Fig. 4 by introducing the predicate $upsorted(A[1:n], k+2)$ to replace: $\forall\ k+2 \leq ind \leq n : A[ind] = maximum\{A[1:ind-1]\}$. Domain abstractions can further abstract the formal annotations with concepts specific to the application domain. The annotation of programs with their assertions using predicate calculus, algebraic specifications, and λ-abstractions are examples of formal specifications. When these specifications use notations that are dependent on the application domain e.g., referring to the average as the GPA in the domain of student grading systems, they become more readable and understandable.

With respect to derivability, writing formal specifications, in general, is not an easy task. It is not as intuitive as in the case of writing natural language specifications. Much more ingenuity and experience are required in order to precisely describe what the component does with a well defined formal notation. The automatic generation of specifications similar to those in Figs. 4 and 5 is a well known problem for the current specifiers and provers[1, 23, 15]. These systems require the user to provide the loop annotations which is an ingenuous task that needs experts' knowledge. The automatic manipulation of formal specifications, however, is better than in the case of informal ones because of their well defined syntax and semantics. In the remainder of this paper, we briefly describe a prototype specifier, CARE-FSQ$_2$, and explain how to enhance it by presenting a knowledge-based approach for generating formal loop specifications.

5 A Component Specification Tool

A prototype specifier has been designed at the University of Maryland as part of the packaging activities performed in the CARE system. It is the second in a series of prototype tools developed under the general name FSQ, for Functional Specification Qualifier[1, 2, 31]. This prototype supports the derivation of programs' specifications and the verification of whether or not the programs meet those specifications. It does not only help to specify and check the partial correctness of finished programs, but it also works on unfinished programs and program fragments. It is a program understanding tool that is based on a formal specification technique.

In a typical session, the user derives the specifications of the program using step-wise abstractions. The user starts by trying to find the correct specification of every program

segment loop in the program as a separate entity. After succeeding in this, the correct specification of the whole program can be found. This methodology of step-wise 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. Currently, CARE-FSQ$_2$ supports a subset of Ada with modifications on the input/output mechanism.

The CARE-FSQ$_2$ prototype helps in checking syntax, static semantics, and generating specifications at the same time. It also provides the capability of carrying out some algebraic simplifications and enables the user to make use of some well defined mathematical functions in the specification of the loop function. However, this prototype does not provide any assistance in a major and difficult task which is annotating the loops with their functions or invariants. To intelligently assist the understanding computer programs, we present a technique that can enhance a specification tool, such as CARE-FSQ$_2$, by mechanically annotating loops.

6 The Analysis of loops

Considerable research has been performed on ways to automatically develop loop invariants (functions). Some of this work centers around heuristic methods which can be used to guide a search for an invariant. The research performed by Dunlop and Basili[12], Katz and Manna[21], Remmers[32], and Wegbreit[37] is representative of these heuristic approaches. Although these heuristic approaches can be helpful in some cases, they offer no guarantees concerning their success. After applying them for a considerable amount of time, you may or may not succeed in finding a correct invariant. Other work focuses on developing algorithmic approaches for finding the invariants (functions) of simple loops. Examples of the later approach can be found in the work of Basu and Misra[7], Dunlop and Basili[11], Katz and Manna[21], Mills[27], Misra[28], and Morris and Wegbreit[29].

A more practical approach which analyzes loops by decomposing them into fragments, was proposed by Waters[36]. The key feature of this analysis method is that it breaks the loop apart in a mechanical way. This decomposition can facilitate both the 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. 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 following section, we introduce a technique that annotates loops with their functional abstractions in a step by step process as depicted in Fig. 6. The analysis of a loop starts by decomposing it into fragments. This decomposition is based on the structural dependencies among the different loop segments. The resulting fragments are analyzed using objects, called plans, stored in a knowledge-base to deduce their functional abstractions. The functional abstraction of the whole loop is then synthesized from the functional ab-

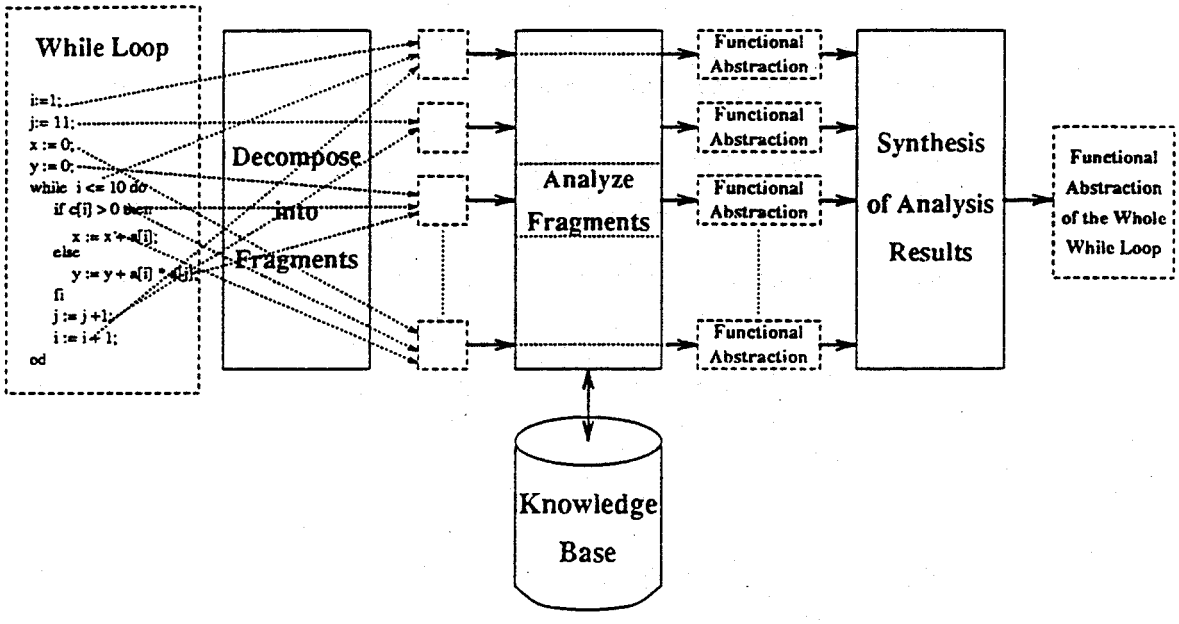

Fig. 6. Overview of the analysis strategy.

stractions of its fragments. Expert knowledge and inference procedures are thus utilized to solve a problem which is regarded formidable to algorithmic approaches.

This approach creates an intelligent assistance to software engineers by adding knowledge to the tool in the form of plans stored in a knowledge-base. It can produce partial documentations if the available knowledge is insufficient for finding complete documentations. This characteristic makes it more flexible and user friendly. Specifically, this approach facilitates the process of deriving loop invariants by mechanically identifying and deriving as many parts of the invariant as possible. The parts of the invariant that cannot be identified, due to an insufficiency of library plans, are linked to the corresponding parts of the loop to serve as a guide for designing future plans suitable for the domain under consideration. Using a logical formalism such as the predicate calculus provides us with the required semantic soundness and expressive power. This is because it is based on a sound mathematical basis[18] and can abstract a wide variety of problems. Furthermore, the formalisms we introduce for knowledge representation are flexible and can be tailored to the needs of different domains. They encapsulate knowledge in a form that can be accessed and reused mechanically. By increasing the number of plans in a system, it can be incrementally developed and enhanced.

The approach we present provides support for maintenance, reuse, and many program development activities such as code reviews, debugging, and some testing approaches. It helps in documenting programs and can be used, along with a theorem prover, in verifying their correctness. It assists the software engineer in rigorously understanding the program under consideration which is especially useful for the various maintenance and reuse activities. By having software components that are very well understood and that are formally specified, we can populate a software repository with well packaged components.

7 A Knowledge-Based Approach for Analyzing loops

We start by introducing the notation we use 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.

7.1 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 number of control variables 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 Fig. 7.

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

The constraints imposed on the condition of a simple while loop are designed such that it is possible to write, in a nonrecursive closed form, the set of values scanned by the control variable. This is because the control variable is modified with fixed values and the values it can scan are bounded by its initial value and the loop condition. 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 it might not be possible to obtain such a specific knowledge because the control variables are not modified with fixed values. Rather, the modifications of the control variables in one iteration are dependent on the behavior of the loop in the previous iterations. Thus, the set of values scanned by the control variables cannot be written in a nonrecursive closed form. For example, while performing a binary search on a sorted array, the modifications of the control variables in one loop iteration are dependent on the array segment currently under consideration which can only be determined based upon the loop behavior in the preceding iterations.

It should be mentioned here that the constraints imposed on the condition of simple while loops restrict them to a class of loops that is a superset of the Pascal *for* loops. The conditions imposed on the set of Pascal *for* loops, F, are stronger since they put a restriction

1.	Number of control variables	Simple loop	General loop
2.	Complexity of loop condition	Noncomposite condition	Composite condition
3.	Complexity of loop body	Flat loop	Nested loop

Fig. 7. The three dimensions used for classifying loops.

on the type of the control variable and the control variable is only altered by unit step[19]. That is why the set of simple while loops, W_1, is a proper superset of F, i.e. $F \subset W_1$.

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 the simple while loops are defined by imposing some restrictions on the general while 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.

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

7.2 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.

More formally, a *Basic Event* (BE) consists of three parts: the condition, the enumeration, and the initialization.

- *The condition* is a logical expression that does not contain the logical operator $\wedge$ and is a part of the loop condition.
- *The enumeration* is a segment that assigns to one or more of the control variables used in the condition the net modification done to them in one loop cycle, if any.
- *The initialization* is a segment that assigns to the control variables modified in the enumeration part their initial values.

An *Augmentation Event* (AE) consists of two parts: the body and the initialization.

- *The body* is a segment that assigns the per-cycle modification, taking place in the loop body, to some variables other than the control variable.
- *The initialization* is a segment that assigns to the same variables modified in the body their initial values.

7.3 Representation of Understanding Knowledge

The information stored in the plan library is divided into two main categories: *Basic Plans (BP's)* and *Augmentation Plans (AP's)*. The basic plans are used to analyze the parts of the loop that control its execution, i.e. BE's. The augmentation plans are used to analyze the other parts in the loop body, i.e. AE's.

The plans correspond to the usual rules utilized in a rule-based system[16]. They are *declarative* representations in which knowledge is specified, but the use to which that knowledge is to be put is not given. To use them they are augmented with techniques that specify what is to be done to the knowledge and how.The plans can be considered as inference rules. When a loop segment matches its antecedent, the rule is fired. The instantiation of the information in the consequent represents the contribution of this plan to the loop assertions.

The structure of the plans is divided into two parts; the antecedent and the consequent. The two plans shown in Figs. 8 and 9 demonstrate the standard structure of the library plans. This structure is suitable for representing the analysis knowledge of different loop classes. This is to avoid having to store duplicate versions of the plans that are used in analyzing different classes. For example, some BP's are used in analyzing BE's of general and simple loops in both inner and outer loop constructs. Some AP's are also used for analyzing both inner and outer loop constructs.

The antecedent of a library plan represents three kinds of knowledge:

- Knowledge about the control variables which is required for the design of the plans' consequents. Even though the control variables can be identified from the **enumeration** of BP's, they cannot be identified from the **body** of AP's. That is why they should be explicitly given in the antecedent of AP's. However, since the individual listing of the control variables can serve to underscore their importance and to facilitate the readability and the comprehension of the plans, they are given in the **control-variables** part of the antecedents of all library plans.

- 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.

- 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:

- Knowledge necessary for the annotation of loops with their Hoare-style[18] specifications. That is why they have the **precondition**, **invariant** and **function** parts where **precondition** and **invariant** have the usual meaning[18]. The **function** part gives

plan-name	DBP_1 (upward-enumeration)
antecedent	
control-variables	$var\#$
initialization	$var_0\#$
condition	$var\#\ R\#\ exp\#$
enumeration	$var\# := SUCC(var\#)$
firing-condition	$R\#$ is relational operator that equals $\leq$ or $<$ $\wedge$
	$var\#$ and $exp\#$ are of a discrete ordinal type.
consequent	
precondition	$var_0\#\ R\#\ exp\#$
invariant	$var_0\# \leq var\#\ R\#\ SUCC(exp\#)$
function	$var\# = SUCC(maximum(\text{final-set}))$
set	$\{ind \mid ind \in T\#\ \wedge\ var_0\# \leq ind \leq PRED(var\#)\}$
final-set	$\{ind \mid ind \in T\#\ \wedge\ var_0\# \leq ind\ R\#\ exp\#\}$
inner-addition	$var_0\# \leq var\#\ R\#\ exp\#$
where,	
$SUCC$	The unary operator that gives the successor of its argument.
$PRED$	The unary operator that gives the predecessor of its argument.

Fig. 8. Example of a basic plan.

plan-name	$SLAP_1$ (summation)
antecedent	
control-variables	$var\#$
body	$cond\# \Longrightarrow lhs\# := lhs\# + exp\#$
initialization	$lhs_0\#$
firing-condition	$lhs\# \neq var\# \wedge lhs\#$ is of a discrete ordinal type and has not been modified by a previous event $\wedge$
	$exp\#$ is an expression that does not include $lhs\#$
consequent	
precondition	$true$
invariant	$lhs\# = lhs_0\# + \sum\{ind \mid ind \in \text{set} \wedge cond\#|_{ind}^{var\#}\}\ exp\#|_{ind}^{var\#}$
function	$lhs\# = lhs_0\# + \sum\{ind \mid ind \in \text{final-set} \wedge cond\#|_{ind}^{var\#}\}\ exp\#|_{ind}^{var\#}$
inner-addition	$lhs\# = lhs_0\# + \sum\{ind \mid ind \in \text{set} \wedge cond\#|_{ind}^{var\#}\}\ exp\#|_{ind}^{var\#}$
where,	
$\sum\{values\}exp$	The summation of exp over the set of $values$.

Fig. 9. Example of an augmentation plan.

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. Furthermore, the **inner-addition** part is needed for the complete annotation of inner loops, if any, in nested constructs.

- In case of simple loops, the constraint imposed on the control variable results in a definite behavior of the loop's control computation. s definite behavior is captured, in the set and final-set parts of the BP's, to produce specifications with compatible

definiteness. These parts give knowledge about the set of values scanned by the control variables at any point during and after the loop execution respectively.

When designing these plans, there is a tradeoff between their complexity and the size of the knowledge-base. Making the plans more abstract can reduce their number in the knowledge-base. However, the increased complexity of a plan complicates its firing process. The example plans presented in this report are designed with a moderate abstraction level. This enables us to easily convey the main ideas behind the analysis technique. The plan in Fig. 8 represents an enumeration construct that goes over a sequence of values of a discrete ordinal type in an ascending order with a unit step. The suffix '#' is used to indicate terms which exist in the antecedent and are not matched with actual values yet or terms which exist in the consequent and are not instantiated with actual values yet. The plan in Fig. 9 represents a construct that performs an accumulation, by summation, of successive values of an expression *exp#* on the variable *lhs#* when the condition *cond#* is satisfied.

When performing analysis of loops in a large domain, the number of plans in the library becomes an important issue. For example, several plans similar to $SLAP_1$, in Fig 9, might be needed to cover other operators(e.g., multiplication and concatenation), data types(e.g., pointers), and special cases (e.g., $exp\# = 1$) of the plan. An increase in the number of library plans leads to a large increase in the search time needed to find the plans which match the events under consideration. To reduce the number of library plans in such cases, they can be designed in a more abstract form. For instance, the plan $SLAP_1$ can be further abstracted by replacing the addition operator, +, with a more abstract one which denotes either addition or multiplication. To further reduce the number of plans, their structure can be made more general. If several similar plans are specializations of a more general case, they can be grouped together in a generalized plan that has a single general antecedent and several consequents organized in a tree structure. At the root is the most general consequent. The children of any node are mutually exclusive specializations of the parent consequent. When the event matches the antecedent of the generalized plan, the search for the appropriate consequent starts at the 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 an additional condition, called **instantiation-condition**, is satisfied. That is, the **firing-condition** must be satisfied in order to select a specific general plan and within the plan the **instantiation-conditions** associated with the edges of the consequents' tree guide the search for the suitable consequent. In limited domains where such abstracted and/or generalized plans are not needed, their inclusion in the library should be avoided because the firing process becomes more complicated and wastes time for no reason.

7.4 The Basic Analysis Strategy

We have argued that the augmentation of a reusable object's interface with the necessary specifications needed for its sound and correct understanding is crucial for reuse. While informal specifications are inherently ambiguous, formal ones can accurately state what is performed by a program segment without having to give unnecessary details. Formal specifications are not adequately utilized mainly because it is hard to automatically generate them. We have developed a tool, CARE-FSQ_2, which provides automatic assistance to

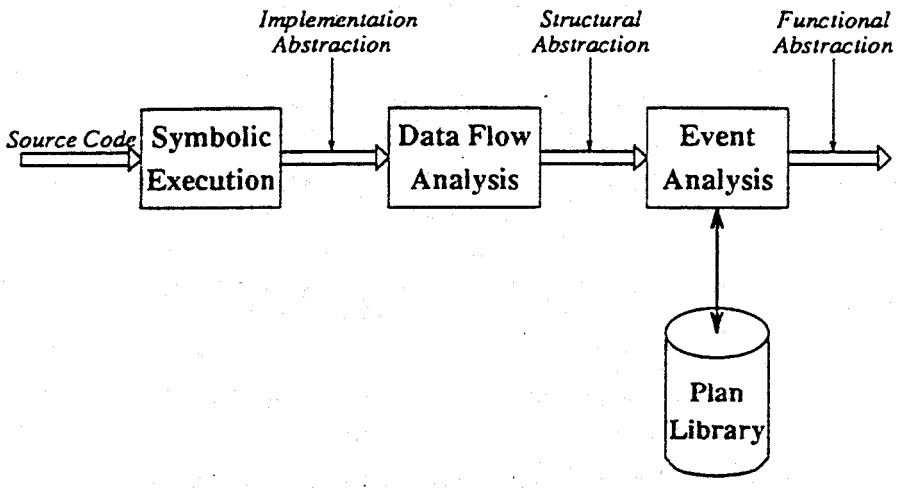

Fig. 10. Analysis of flat simple loop structures.

software engineers in generating the formal specifications of programs[1, 2]. Such a specifier, however, does not assist the ingenious task of annotating loops with their specifications. That is why we have designed analysis techniques which can provide mechanical assistance to the generation of formal specifications of different loop classes in many domains. We have applied these analysis techniques to loops in the domain of basic algorithmic structures and are in the process of applying them to the domain of scheduling university courses. 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.

In this paper, 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 assist the generation of formal specifications which enhance the reusability of available code components in a specific domain of interest.

The annotation of flat simple loops having noncomposite conditions with their functional abstractions is performed in a step by step process utilizing several techniques as depicted in Fig. 10.

The first analysis phase, which is the symbolic execution of the loop body, abstracts away a program's language and implementation specific features. Hence it 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 basic and augmentation events. This analysis reveals the structure of the loop and maps its implementation abstraction to a structural abstraction.

Finally, an analysis of basic and augmentation events, using a plan library, transforms the structural abstraction into a functional abstraction of the loop. The functional abstraction reveals the logical, as opposed to the syntactical or structural, details of the loop using predicate calculus assertions.

The details of the analysis method are explained below. The descriptions of the analysis steps are interspersed with their application on the following example. We have chosen a loop construct of moderate complexity that helps in explaining the basic ideas behind our approach within the appropriate space limitation. However, we have designed plans which cover some basic algorithmic structures such as enumeration, value accumulation, sorting, and searching.

Example

```
i, j, x, y: integer;
a: array[1..20] of integer;
c: array[1..10] of integer;
                .
                .
i := 1;
j := 10;
x := 0;
y := 0;
while i <= 10 do
    j := j + 1;
    if c[i] > 0 then
        x := x + a[i];
    else
        y := y + a[i] * a[j];
    fi
    i := i + 1;
od
```

This loop can be classified as a simple loop since the variable i is a unique control variable. In addition, its modification is not dependent of the values of any other variables modified within the loop.

The first analysis phase symbolically executes the body of the loop. The purpose of this symbolic execution is to remove the coupling between the statements in the loop body and to make the variables' new values dependent only on the values resulting from the last iteration of the loop. Applying this analysis phase on the previous loop yields the following set of statements which are assumed to be simultaneously executed as in a concurrent assignment. These statements represent an implementation abstraction of the loop body which eliminates a program's language and implementation specific features.

$j := j + 1,$
$i := i + 1,$
$c[i] > 0 \Longrightarrow x := x + a[i],$
$c[i] \leq 0 \Longrightarrow y := y + a[i] * a[j + 1]$

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 a basic and augmentation events[36]. That is why the output of this phase is called a structural abstraction

of the loop.

The BE of the loop is constructed by letting the loop condition, B, be the condition of the BE. 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, is placed before the loop and provides the initial value of the control variable.

The symbolic execution result is decomposed into AE's by first removing the enumeration part of the basic loop. Then, the minimal subsegments of code which do not have data flow going to other parts of the loop body are recursively identified and isolated. Whenever two, or more, independent segments are identified, they are isolated in a random order. The resulting segments are ordered such that the ones identified first are analyzed last. The initialization, if any, is placed before the loop and provides the initial values of the variables modified in the augmentation body.

The application of the second analysis phase to our example yields four events. Since the segment that modifies x is independent on the segments that modify j and y, its relative order with respect to them is irrelevant. However, any of the resulting orders of the AE's should satisfy the condition that the event which modifieds y is ordered after the event which modifies j. This is because the modification of y is dependent on the way j is modified. Thus, the ordering of the AE's makes it possible to propagate the effect of analyzing an event to the analysis of other events dependent on it. The resulting ordered events are as follows:

1. The basic event
   condition: $i \leq 10$
   enumeration: $i := i + 1;$
   initialization: $i := 1;$

2. Augmentation event 1
   body: $j := j + 1;$
   initialization: $j := 10;$

3. Augmentation event 2
   body: $c[i] > 0 \Longrightarrow x := x + a[i];$
   initialization: $x := 0;$

4. Augmentation event 3
   body: $c[i] \leq 0 \Longrightarrow y := y + a[i] * a[j + 1];$
   initialization: $y := 0;$

Finally, we try to match the loop events with the antecedents of the plans stored in the knowledge-base. The result of matching each event is the name of the plan matched along with the unification of the terms in the event with the ones in the plan. More formally, the *Analysis Knowledge* (AK) of a variable modified by a certain event is the result of matching this event with the antecedent of a specific library plan. It is represented as a n-tuple where n is dependent on the specific matched plan. The first term of the tuple is the name of the matched plan. The remaining (n-1) terms are the unification of the # variables with the actual ones in the event.

The instantiation of the consequents of the matched plans with the actual variables 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

program events with the antecedents of the plans shown in Figs. 8 and 9 respectively. The analysis knowledge of the variables modified in the events is as follows:

1. $AK(i) = (DBP_1,\ var\# : i,\ T\# : integer,\ var_0\# : 1,\ R\# :\leq,\ exp\# : 10)$
2. $AK(j) = (SLAP_1,\ var\# : i,\ cond\# : true,\ lhs\# : j,\ exp\# : 1,\ lhs_0\# : 10)$
3. $AK(x) = (SLAP_1,\ var\# : i,\ cond\# : c[i] > 0,\ lhs\# : x,\ exp\# : a[i],\ lhs_0\# : 0)$
4. $AK(y) = (SLAP_1,\ var\# : i,\ cond\# : c[i] \leq 0,\ lhs\# : y,\ exp\# : a[i] * a[j+1],\ lhs_0\# : 0)$

Instantiating the consequents of the identified plans with the actual variables yields the following results:

1. **precondition** $1 \leq 10$
   invariant $1 \leq i \leq 11$
   function $i = SUCC(maximum(\{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq 10\}))$
   set $\{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq i-1\}$
   final-set $\{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq 10\}$
   inner-addition $1 \leq i \leq 10$
2. **precondition** $true$
   invariant $j = 10 + \sum\{ind_1 \mid ind_1 \in \{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq i-1\} \wedge true\}\ 1$
   function $j = 10 + \sum\{ind_1 \mid ind_1 \in \{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq 10\} \wedge true\}\ 1$
   inner-addition $j = 10 + \sum\{ind_1 \mid ind_1 \in \{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq i-1\} \wedge true\}\ 1$
3. **precondition** $true$
   invariant $x = 0 + \sum\{ind_2 \mid ind_2 \in \{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq i-1\} \wedge c[ind_2] > 0\}\ a[ind_2]$
   function $x = 0 + \sum\{ind_2 \mid ind_2 \in \{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq 10\} \wedge c[ind_2] > 0\}\ a[ind_2]$
   inner-addition $x = 0 + \sum\{ind_2 \mid ind_2 \in \{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq i-1\} \wedge c[ind_2] > 0\}\ a[ind_2]$

Since j has been modified before and is responsible for the data flow into the fourth augmentation, the instantiation is done with j_{pred} instead of j. This notation denotes the value of j as deduced from the invariant of the loop under consideration. It is used to distinguish the variables which have been changed before from those which have not. It is obvious from this step how the ordering of AE's affects the analysis by taking into consideration the modification of the variable j when analyzing other AE's that depend on it.

4. **precondition** $true$
   invariant $y = 0 + \sum\{ind_3 \mid ind_3 \in \{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq i-1\} \wedge c[ind_3] \leq 0\}\ (a[i] * a[j_{pred}+1])|^{i}_{ind_3}$
   function $y = 0 + \sum\{ind_3 \mid ind_3 \in \{ind \mid ind \in integer\ \wedge\ 1 \leq ind \leq 10\} \wedge c[ind_3] \leq 0\}\ (a[i] * a[j_{pred}+1])|^{i}_{ind_3}$

inner-addition $y = 0 + \sum\{ind_3 \mid ind_3 \in \{ind \mid ind \in integer \ \wedge \ 1 \leq ind \leq i-1\} \wedge c[ind_3] \leq 0\} \ (a[i] * a[j_{pred} + 1])|_{ind_3}^{i}$

The functional abstraction of the loop is synthesized from the instantiated plans' consequents. The precondition, invariant, and function are constructed by taking the conjunction of the corresponding parts in the instantiated consequents. For instance, the precondition and invariant of our example have the following form:

$$\text{Precondition: } 1 \leq 10$$

$$\begin{gathered} \textit{Invariant: } (1 \leq i \leq 11) \wedge \\ j = 10 + \sum\{ind_1 \mid ind_1 \in \{ind \mid ind \in integer \ \wedge \ 1 \leq ind \leq i-1\}\} \ 1 \ \wedge \\ x = \sum\{ind_2 \mid ind_2 \in \{ind \mid ind \in integer \ \wedge \ 1 \leq ind \leq i-1\} \ \wedge \ c[ind_2] > 0\} \ a[ind_2] \ \wedge \\ y = \sum\{ind_3 \mid ind_3 \in \{ind \mid ind \in integer \ \wedge \ 1 \leq ind \leq i-1\} \wedge c[ind_3] \leq 0\} \\ (a[i] * a[j_{pred} + 1])|_{ind_3}^{i} \end{gathered}$$

The disadvantage of such general and domain independent functional specifications is that they are especially hard to read when the components become more complicated. To improve the readability of such specifications, some of the commonly occurring predicates can be replaced with a more intuitive and easy to read notation. This more intuitive notation can be in terms of a widely known or a domain specific concept. For example, assume that in some domain a manufacturer produces 10 different products. For each product, there are two alternative quantities which can be manufactured. The first alternative produces a standard quantity of product i as given by $a[i]$. The second alternative produces a higher quantity that is a fixed multiple of the standard one. The fixed multiplier of product i is given in the location of the array a pointed to by the *high-index*: $i + 10$. Based on market studies, it is determined whether to produce the quantity of each product according to alternative 1, if $c[i] > 0$, or alternative 2, if $c[i] \leq 0$. The total quantity of products produced according to alternatives 1 and 2 are called *standard-quantity* and *large-quantity* respectively. The above invariant can thus be replaced with a domain specific one that is more readable and understandable, in this domain, as follows:

$$\begin{gathered} (1 \leq i \leq 11) \wedge \\ j \ = \textit{high-index} \ (i) \ \wedge \\ x \ = \textit{standard-quantity} \ (a, \ 1 \ .. \ i-1) \ \wedge \\ y \ = \textit{large-quantity} \ (a, \ 1 \ .. \ i-1) \end{gathered}$$

In addition to improving the readability of the specifications, such replacements do not affect the semantic soundness of the specifications. This is because each new term still has an underlying rigorous definition that can be utilized if desired. These replacement can be done explicitly as performed here by producing the functional specifications and then the more abstract ones. Otherwise, they can be implicitly performed by designing the plans such that their consequent are directly written in terms of the more abstract terms. In the former case the library 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.

The advantage of the resulting specifications is that they explain the function of the

algorithm in concise terms that do not introduce any ambiguities. They explain what the algorithm does without giving the unnecessary information of how it does it. Hence, such specifications can help the reuser of a component in understanding it which, in turn, is a requisite for its correct modification. The clear and concise description of the component function can also assist in the identification of the component whether this identification is performed mechanically or not.

7.5 Evolution of the Knowledge-Base

The success of the developed analysis techniques in a specific domain is dependent, to a great extent, on the efficient and correct design of the plans. The resulting specifications are as accurate, readable, and correct as the plans are. That is why the task of designing the plans and managing the knowledge-base for a specific domain of interest should be performed by an expert in both the desired domain and formal specifications.

To develop the knowledge-base, the desired domain should be analyzed to design an initial set of plans which is 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 where the knowledge-base manager needs to closely monitor its utilization.

While using the knowledge-base, a failure to identify the specification of a loop segment indicates that either the segment 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 segment to see if he/she can modify it. If no error is detected, 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 abstract and/or generalize existing plan(s) to cover the new case.

Determining the appropriate generalization and abstraction level of the plans is largely dependent on the domain under consideration. The basic understanding of a domain is represented in the initial set of plans constituting the knowledge-base. Further utilization of the knowledge-base is apt to reveal inadequacies in it with respect to the sufficient number of plans, the level of abstraction, and the ease of instantiation. This utilization is also likely to improve the understanding of the domain and increase the knowledge of its details. In such cases, there is a tradeoff between adding new plans or generalizing and abstracting existing ones. On one hand, the generalization and abstraction of the plans reduces their number and thus can lead to a reduction in the search time needed to find the required ones. It also reduces the space needed to store them. On the other hand, it complicates the pattern matching and unification performed on them which can lead to an increase in the search time. It also complicates the instantiation of the proper consequent. Further research needs to be performed to choose the level of abstraction and generalization of the plans. The suitable level can be achieved by experimentation and by studying the specific domain.

The controlled utilization of the knowledge-base serves to adapt the plans and make their abstraction level, number, and naming conventions suitable for the domain under

consideration. Any domain specific abstractions and specifications which are felt to be inappropriate are also reported, by the users, to the knowledge-base manager(s). When enough consensus is reached, the inappropriate abstractions and specifications are replaced with more suitable ones.

8 Conclusion

We have discussed the importance of properly packaging reusable components before adding them to a software repository. The packaging of candidate reusable components can enhance their reusability by facilitating their identification, their understanding, and their modification during the actual reuse process.

One of the requisites for the success of the reuse process is being able to understand the component being reused. This can be achieved by augmenting it, during packaging, with the necessary specifications needed for its sound and correct understanding. These specifications can also assist in its identification and modification. That is why we have focused on a knowledge-based approach for annotating reusable components using formal specifications. We have demonstrated how one can produce abstract specifications which are readable and have an underlying formal basis at the same time. Even though we have designed techniques for specifying programs with different loop classes, we only described the basic analysis strategy. This description served to explain how to produce specifications that can enhance the reusability of software components.

Acknowledgement

We would like to thank Gianluigi Caldiera and Dieter Rombach for their helpful contributions to a variety of aspects presented in this paper. This research 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 at College Park,1990.

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

[3] J. W. Bailey and V. R. Basili, "Software Reclamation: Improving Post-Development Reusability", Proceedings 8th Annual Conference on Ada Technology, Atlanta, GA, March 1990.

[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] V. R. Basili, "Software Development: A Paradigm for the Future", *Proceedings of the 13th Annual International Computer Software and Applications Conference*, Orlando, Florida, Sept. 1989, pp. 471-485.

[6] V. R. Basili and H. D. Rombach, "Support for Comprehensive Reuse", Software Engineering Journal, vol. 6, no. 5, Sept. 1991, pp. 303-316.

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

[8] T. Biggerstaff and C. Richter, "Reusability Framework, Assessment, and Directions," *IEEE Software*, March 1987, pp. 41-49.

[9] G. Booch, "Software Components with Ada Structures, Tools, and Subsystems", The Benjamin/Cummings Publishing Company, 1987.

[10] G. Caldiera and V. R. Basili, "Identifying and Qualifying Reusable Software Components", *IEEE Computer*, Feb. 1991, pp. 61-70.

[11] D. D. Dunlop and V. R. Basili, "A Comparative Analysis of Functional Correctness," *Computing Surveys*, vol. 14, no. 2, June 1982, pp. 229-244.

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

[13] R. B. France and V. R. Basili, "A Pattern-Driven Approach to Code Analysis for Reuse," Technical Report UMIACS-TR-91-159, CS-TR-2802, Department of Computer Science, University of Maryland, College Park, MD 20742, Nov. 1991.

[14] J. A. Goguen, "Parameterized Programming", *IEEE Trans. on Software Engineering*, vol. SE-10, no. 5, Sept. 1984, pp. 528-543.

[15] D. Good *et al*, "Mechanical Proofs about Computer Programs", in C. A. R. Hoare and J. C. Shepherdson: "Mathematical Logic and Programming Languages", Prentice-Hall International, 1985, p. 55-75.

[16] F. Hayes-Roth, "Rule-Based Systems," *Communications of the ACM*, vol. 28, no. 9, Sept. 1985, pp. 921-932.

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

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

[19] K. Jensen and N. Wirth, "Pascal User Manual and Report," Springer-Verlag, 1985.

[20] B. Joo, "Adaptation and Composition of Program Components", Ph.D. Dissertation, Department of Computer Science, University of Maryland, College Park, Maryland, 1990.

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

[22] S. Katz, C. A. Richter, and Khe-Sing The, "Paris: A System for Reusing Partially Interpreted Schemas", Nineth International Conference on Software Engineering, March 1987.

[23] R. A. Kemmerer and S. T. Eckmann, "UNISEX: A UNIx-based Symbolic EXecutor for Pascal," *Software Practice and Experience*, vol. 15, no. 3. May 1985, pp. 439-458.

[24] R. G. Lanergan and C. A. Grasso, "Software Engineering with Reusable Design and Code", *IEEE Trans. on Software Engineering*, vol. SE-10, no. 5, Sept. 1984, pp. 498-501.

[25] S. Letovsky, "Program Understanding with the Lambda Calculus," *Proceedings of the 10th International Joint Conference on AI*, Aug. 1987, pp. 512-514.

[26] M. D. Lubars and M. T. Harandi, "Addressing Software Reuse Through Knowledge-Based Design", *Software Reusability, edited by T. J. Biggerstaff and A. J. Perlis*, vol. II, chapter 16, ACM Press, 1989.

[27] H. D. Mills, "The New Math of Computer Programming," *Communications of ACM*, vol. 18, no. 1, Jan. 1975, pp. 43-48.

[28] J. Misra, "Some Aspects of the Verification of Loop Computations," *IEEE Trans. on Software Engineering*, vol. SE-4, no. 6, Nov. 1978, pp. 478-486.

[29] J. H. Morris Jr. and B. Wegbreit, "Subgoal Induction," *Communications of ACM*, vol. 20, no. 4, April 1977, pp. 209-222.

[30] R. Prieto-Diaz and P. Freeman, "Classifying Software for Reusability", *IEEE Software*, Jan. 1987, pp. 6-16.

[31] S. S. Qian, "A Tool for Understanding Software Components," Master's thesis, University of Maryland at College Park,1989.

[32] J. H. Remmers. "A Technique for Developing Loop Invariants," *Information Processing Letters*, vol. 18, 1984, pp. 137-139.

[33] H. B. Reubenstein and R. C. Waters, "The Requirements Apprentice: Automated Assistance for Requirements Acquisition", *IEEE Trans. on Software Engineering*, vol. 17, no. 3, pp. 226-240,

[34] C. Rich and R. C. Waters, "Formalizing Reusable Software Components in the Programmer's Apprentice", *Software Reusability, edited by T. J. Biggerstaff and A. J. Perlis*, vol. II, chapter 15, ACM Press, 1989.

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

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

[37] B. Wegbreit, "The Synthesis of Loop Predicates," *Communications of ACM*, vol. 17, no. 2, Feb. 1974, pp. 102-112.