Towards Automated Support for Extraction of Reusable Components

S. K. Abd-El-Hafiz V. R. Basili G. Caldiera
Institute for Advanced Computer Studies,
Department of Computer Science,
University of Maryland, College Park, MD 20742, U.S.A.

Abstract

A cost effective introduction of software reuse techniques requires the reuse of existing software developed in many cases without aiming at reusability. This paper discusses the problems related to the analysis and reengineering of existing software in order to reuse it. We introduce a process model for component extraction and focus on the problem of analyzing and qualifying software components which are candidates for reuse. A prototype tool for supporting the extraction of reusable components is presented. One of the components of this tool aids in understanding programs and is based on the functional model of correctness. It can assist software engineers in the process of finding correct formal specifications for programs. A detailed description of this component and an example to demonstrate a possible operational scenario are given.

1 Introduction

Successful reuse of software resources can increase the overall quality and productivity in software projects by a large factor. Some of the problems that still limit software reuse are:

1. The difficulty of understanding a given software product in the absence of its original developers.
2. The scarce availability of reusable objects, even though there is a tremendous amount of available software.
3. The difficulty of retrieving, from a large data base, software components which can best match the given semantics requirements.
4. The lack of extraction and adaptation techniques that facilitate the reuse process.

New process models for software development should substitute the existing ones that are not defined to benefit from or support reuse. These new models should take advantage of reuse, introduce more reusable resources, and overcome the existing problems that limit reuse.

Developing reusable components is generally more expensive than developing specialized code, because of the overhead of designing for reusability and maintaining the component repository. A rich and well-organized catalog of reusable components is the key to a successful component repository and a long term economic gain. Moreover, such a catalog will not be available to an organization unless 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[1]. A technique for extracting reusable components can improve productivity since it provides the software developer with components that are ready for reuse or need minor adaptation. Moreover, it can improve the software quality as it helps in better understanding these components during the extraction process.

In this paper, we use a process model[2] that serves not only to enhance the development of the project under consideration but also to organize and plan for better reuse technology in future projects. This model splits the traditional life-cycle model into two separate organizations, the project organization and the experience factory. In this framework we introduce a process model for component extraction and focus on the problem of qualifying candidate software components for reuse.

A prototype tool constituting one of the elements of an integrated system for extracting reusable compo-

nents is described. This prototype tool helps in understanding programs by deriving their specifications and is based on the functional model of correctness[3, 4]. The tool could be applied to program fragments as well as to complete programs and it helps in simultaneously checking syntax, static semantics, and generating specifications. We conclude the paper with an example to demonstrate a possible operational scenario of the tool.

2 Organizing the component extraction

Currently, all reuse occurs in the project development, where there is a completion deadline and the top priority is to deliver the system on time. This makes the objective of developing reusable software, at best, a secondary concern. Besides, project personnel cannot recognize the pieces of software appropriate for other projects.

We make use of a reuse-oriented model based on two separate organizations[2]:

- **The project organization:** Its goal is to deliver the systems required by the customer. The process model can be chosen based upon the characteristics of the application domain, taking advantage of prior software products and experience.

- **The experience factory:** It supports project development by analyzing and synthesizing all kinds of experience, acting as a repository for such experience, and supplying that experience to various projects on demand. Within the experience factory, we can identify various sub-organizations. One of them is the component factory which develops reusable components, extracts reusable components from existing systems, and generalizes or remodels any previously produced component.

Different conceptual architectures can be used for the component factory[5]. At one extreme there is the clustered architecture in which all software development activities are concentrated in the project organization and the component factory is dedicated only to processing already existing software. At the other extreme there is the detached architecture in which the development activities are concentrated in the component factory and the project organization performs only high-level design and integration. The clustered architecture is much closer to the way software is currently implemented. The development of the components is probably faster in the project organization since there is less communication overhead and more direct pressure for their delivery. On the other hand, the components developed are more context dependent. In the detached architecture, there is more emphasis on developing general purpose components in order to serve several project organizations more efficiently. On the other hand, there are more chances for bottlenecks and for periods of inactivity due to the lack of requests from the projects. The detached architecture is probably better suited for environments where the practice of reuse is formalized and mature. An organization that is just starting with reuse should probably instantiate its component factory using the clustered architecture and then, when it reaches a sufficient level of maturity and improvement with this architecture, start implementing the detached architecture in order to continue the improvement.

In any case, the extraction of reusable components is a characteristic activity of the component factory. The next section will present in detail the features of this activity, in the framework of a component factory. Caldiera and Basili[6] have proposed a process model for the extraction of reusable components in two phases: the identification phase and the qualification phase (see figure 1). The necessary human intervention in the second phase is the main reason for splitting the process in two steps. The first phase, which can be fully automated, reduces the amount of expensive human analysis needed in the second phase by limiting analysis only to components that really look worth considering.

3 The extraction process

3.1 Identification

Program units are automatically extracted and made to be independent compilation units. These independent units are measured according to observable properties related to their potential for reuse in three steps. These steps are summarized here:

1. Definition of the reusability attribute model: A set of automatable measures that captures the characteristics of potentially reusable components is defined along with acceptable ranges of values for these metrics.

2. Extraction of components: Modular units (e.g. C functions. Ada subprograms or blocks, or Fortran subroutines) are extracted from existing software and

completed so that they have all the external references needed to reuse them independently.

3. Application of the model: The current reusability attribute model is applied to the extracted, completed components. Components whose measures are within the model's range of acceptable values become candidate reusable components to be analyzed in the qualification phase.

A detailed description of the component identification phase, a definition of a basic reusability attribute model, and an application of this model on several case studies using a computer-based system have already been discussed in the literature[6].

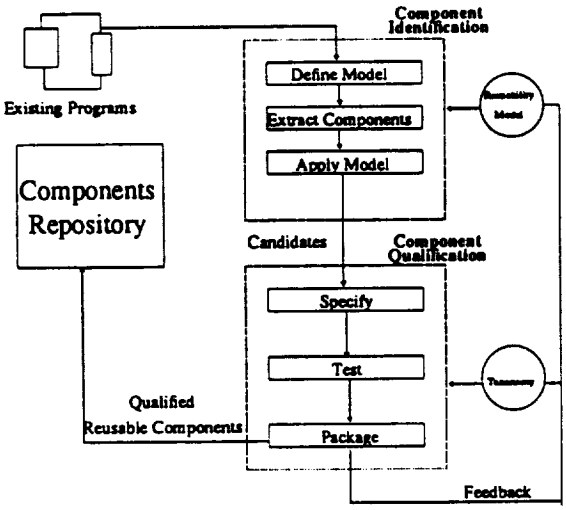

Figure 1: Component extraction.

3.2 Qualification

The extracted components are analyzed in order to understand them and record their meaning. The components are packaged by associating with them a reuse specification, a significant set of test cases, a set of attributes based on a reuse classification scheme, and a set of procedures for reusing the component. This phase consists of following steps:

1. Formal specification: A precise description of what the component does is generated and some assurance is obtained that the component meets the requirements.

Since formal specifications are based on mathematical notations, they help in understanding the software by removing the ambiguities which might be introduced by any informal notation. Formal specifications are different from the programs they specify since they only express the behavior of the program without stating how the program derives this behavior. So, formal specifications are the basis for selecting and storing software components as they improve understandability and assist in producing more reliable and higher quality software. Since the specification of complex tasks may in itself be complex, the process of specification construction must be formalized and supported by automated tools. In the next section, we will describe a prototype tool that aids in understanding programs. This tool provides automated support for deriving the functional specifications of programs and proving their partial correctness. In other words, it helps in proving that the program is consistent with its specification but does not prove its termination.

Formally specifying a software component and proving its partial correctness do not mean that the component will pass this step. There are several other properties that should exist in the candidate components for the sake of understandability. We must not ignore other important features such as proper documentation, use of meaningful variable names, and the structured style of programming. The informal information that the software engineer deals with cannot be ignored relying on the fact that the automated specifications tools will supplement those features. The informal information is important in explaining some intuitive ideas that are hard to explain using formal specifications.

Since we need both formal and informal information, a domain expert is needed to perform the specification step. This expert extracts the formal specification of each candidate reusable component, assisted by the automated tools available, and examines the other informal features that cannot be judged using automated tools. Components that are not relevant, not correct, or whose functional specification is not easy to extract are discarded. The expert reports reasons for discarding candidates and other insights that will be used to improve the reusability attributes model.

2. Testing: Test cases are generated, executed and associated with components. Deriving the functional specification and proving the correctness of a program do not mean that it will not fail when compiled and/or executed. This might simply be due to the fact that termination of the program has not been proven. Moreover, in most verification and specification systems, arithmetic operations ignore things such as overflow, underflow, and round-off errors.

Testing can take advantage of the functional specification generated by performing functional testing. Also, structural testing can be done using a coverage analyzer. If, as is likely, the component needs a

'wrapping' to be executed, the testing step generates this wrapping. If a component passes the testing then test cases, wrapping, and test results are stored in the component repository. Components that do not satisfy the test are discarded. Again, the reasons for discarding candidates are recorded and used to improve the reusability attributes model and possibly the process for extracting the functional specification. This is most likely the last step at which a component will be discarded.

3. Packaging: The extracted candidates are stored in the component repository along with their functional specifications and test cases. The component repository is actually a data base of experience in which information on software products, processes, and measures of aspects of them is stored. That is why we organize this data base by classifying both the reusable components and their development histories according to several domain dependent criteria.

Information for the future reuser is provided in a manual that contains a description of the component's function and interfaces as identified during generation of its functional specification, directions on how to install and use it, information about its procurement and support, and information for component maintenance.

At the end of each process cycle the reusability attribute model is updated by drawing on information from the qualification phase to add more measures, modify or remove measures that proved ineffective, or alter the range of acceptable values. This step requires analysis and possibly even further experimentation. The taxonomy is updated by adding new attributes or modifying the existing ones according to problems reported by the experts who classify the components.

4 The CARE system

4.1 Overview

The CARE[6] system(CARE[1]: Computer Aided Reuse Engineering) has been designed to support the proposed process model for extracting reusable components. As shown in figure 2, it consists of two main subparts: the component identifier and the component qualifier. The component identifier consists of the model editor, which helps in defining and modifying the reusability attributes model, and the component extractor which applies such model to the programs. The component qualifier consists of the specifier, the tester, and the packager. The current version of the CARE system consists of the component extractor and the specifier. It runs on a Sun Workstation and supports ANSI C and Ada. In the rest of this section we focus on the description of the specifier.

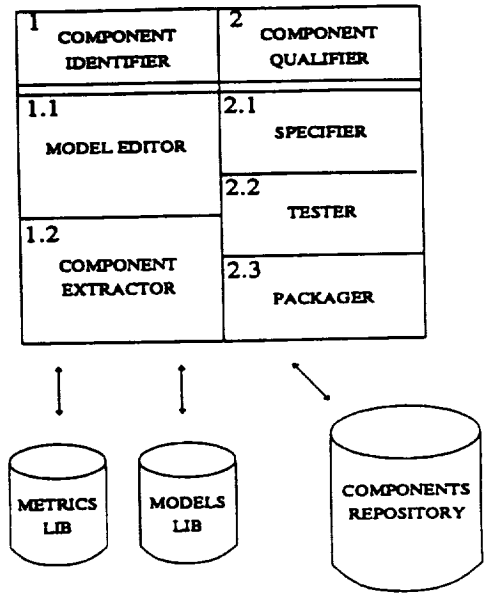

Figure 2: CARE system architecture.

4.2 The component specification tool

The prototype specifier included in the CARE tool is the second in a series of prototype tools developed at the Computer Science Department of the University of Maryland under the general name FSQ, for Functional Specification Qualifier. 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. CARE-FSQ$_2$ uses Mills' functional model of correctness[3, 4] in order to derive the specifications. This model requires the user to provide only the loop function and then a technique is provided to derive the program specification. Other techniques[7, 8] require the user to provide an entry assertion, an exit assertion, and a loop assertion. Those techniques are more useful in verifying that the program is consistent with its specification. The process of deriving specifications helps more in understanding the software. Moreover, the functional method pro-

[1]The CARE system is under development at the Computer Science Department of the University of Maryland

vides simple and intuitive notations that can be easily understood.

The CARE-FSQ$_2$ prototype helps in checking syntax, static semantics, and generating specifications at the same time. CARE-FSQ$_2$ 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.

4.2.1 Formal foundation: Each statement S is given a meaning as a function from a program state to another state. A state is a mapping from the variable names to their current values. The square bracket notation is used to denote the function represented by the program construct contained inside the brackets, i.e. $[S]$ represents the function computed by the statement S. We use four basic structures[3, 4]:

1. Assignment

The meaning of the assignment $v := e$, where v is a variable and e is an expression, is:

$$[v := e] = \{(S,T) : T = S \text{ except that } [v](T) = [e](T)\}$$

We can define the meaning of variables and expressions as a mapping from a state to a value.

2. Composition

If A and B are statements and $\circ$ is functional composition, we have:

$$[A;B] = [A] \circ [B]$$

3. Alternation

$$[\ if\ B\ then\ S\ fi\ ] = \{(U,[S]U) : [B](U) = true\} \cup \{(U,U) : [B](U) = false\}$$

$$[if\ B\ then\ S_1\ else\ S_2\ fi] = \{(U,[S_1]U) : [B](U) = true\} \cup (U,[S_2]U) : [B](U) = false\}$$

4. Iteration

$$[while\ B\ do\ S\ od] = \{(T,U) : \exists k \geq 0 : \forall 0 \leq i < k\ (([B]([S]^i(T)) = true \wedge [B]([S]^k(T)) = false \wedge [S]^k(T) = U)\}$$

In other words, the loop function is undefined for a state T unless there is a natural number k which denotes the number of iterations after which the test first fails. T is then transformed to the k-fold composition of S on T. In order to carry out practical proofs, the following characterizing theorem is needed[9].

Theorem

Let W be the program fragment *while B do S od*, Then $f = [W]$ if and only if:

1. $domain(f) = domain([W])$
2. $([B](T) = false) \implies f(T) = T$
3. $f = [if\ B\ then\ S\ fi] \circ f$

This theorem provides a method for deriving the correct loop function f:

1. Guess or work out a trial function f.
2. Use the three conditions of the theorem to check that the trial function is correct.

A trace table can be used to organize the derivation of program meanings (by a symbolic execution of the program)[4, 9].

The strength and weakness of the functional method, in comparison with other specification techniques, originate from the fact that even though exact functions state accurately the meaning of a loop, they are harder to work with than the weak assertions that suffice when there is a loop initialization providing a precondition.

4.2.2 The implementation: CARE-FSQ$_2$ is implemented using the Synthesizer Generator[10] and Maple, an interactive algebraic symbolic executor[11]. An overview of the tool is shown in figure 3. The Synthesizer Generator requires as an input a description of an attribute grammar and generates from it a hybrid language-based editor that allows a combination of text editing and structure editing. As the user edits program text and annotations, the system creates and edits abstract syntax trees that represent pieces of programs and their specifications. The attributes of the nodes of this tree carry information about the static semantics of the program as well as its specifications, and they are evaluated incrementally. The basic feature of Maple is its ability to simplify expressions involving unevaluated elements. As each complete statement is entered by the user, it is evaluated and the results are printed on the output device. Maple enables carrying out algebraic simplifications during the symbolic execution. In order to overcome the limitations of Maple in the evaluation of boolean expressions, CARE-FSQ$_2$ has an interactive feature that allows the user, before writing the specifications, to simplify boolean expressions and the expressions containing array notations.

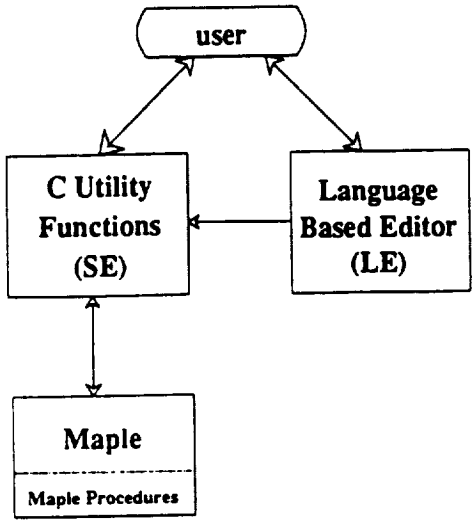

Figure 3: Overview of CARE-FSQ$_2$.

In a typical CARE-FSQ$_2$ session, the user derives the specifications of the program using step-wise abstractions. In other words, the user starts by trying to find the correct specification of every 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 data types supported are integer, boolean, character, a restricted form of floating point, constrained arrays, and user defined data types. The basic control structures of Ada are supported except unconditional 'go to' statements, and case statements. Static semantic checking is also included. A brief description of the input/output mechanism and the specification language is given in the rest of this subsection.

Input and output is done through atomic and stream ports[12]. A subprogram, called an elementary process, accepts input data from input ports, performs computation specified with an Ada-like notation, and returns results through output ports. The input and output of single data items can be carried out through atomic ports. Stream ports are used as schemes for data types whose elements can be accessed in a linear order. The stream ports of one process can be bound to particular data types to produce the implementation. Input and output ports can be bound to files to communicate with the system. This form of data abstraction helps in making the specification process more general and easier. The following seven operations are defined for atomic and stream ports:

1. Receive(p): To Receive a value via the input port p from the source associated with the port.
2. Send(p): To Send a value via the output port p to the destination associated with the port.
3. Initialize(p): To open the stream associated with the stream port p for reading.
4. Receive(p, v): To receive a value into a variable v from the stream associated with the input port p.
5. Send(p, v): To send the value of variable v to the stream associated with the output port p.
6. isEOS(p): A boolean function to check if end of stream is reached in the input stream port p.
7. Finalize(p): To close the stream associated with the port p. The effect of finalization for an output stream port is that the function isEOS becomes true at the consumer process.

The specifications for CARE-FSQ$_2$ are written using guarded command sets whose syntax is:

< *guarded command set* > ::=
    < *guarded command* >
    { | < *guarded command* >}

< *guarded command* > ::=
    < *boolean expr* > →
    < *concurrent assignment* >

< *concurrent assignment* > ::=
    < *var* > := < *expr* > | < *var* > ,
    < *concurrent assignment* > , < *expr* >

A concurrent assignment is an extension of the assignment statement where a number of different variables can be substituted simultaneously. The concurrent assignment statement is denoted by a list of different variables to be substituted at the left hand side of the assignment operator and an equally long list of expressions as its right hand side. The ith variable from the left hand list is to be replaced by the ith expression from the right hand list. The expressions can include calls to some mathematical functions such as min, max, product, sum, factorial, igcd (greatest common divisor), irem (remainder), and iquo (quotient).

An array is considered to be a partial function from subscript values to the type of array elements. The command $a(i) := e$ assigns a new function to a, a function that is the same as the old one except that at the argument i its value is e. The notation (a, i, e) is used to denote the array that is the same as a except when applied to the value i yields e. The notation $(a, index = m..n, e)$ is used to denote the array that is the same as a except when applied to index values between m and n, i.e. $m \leq index \leq n$, it yields e. The expression e can be a function of the bound variable $index$. To make the two notations consistent, (a, i, e) is written $(a, index = i, e)$ where $index$ is a bound variable. The notation defined for arrays are used for stream ports as well. A stream port is treated as an array whose subscript is of type integer with the first element subscript being one.

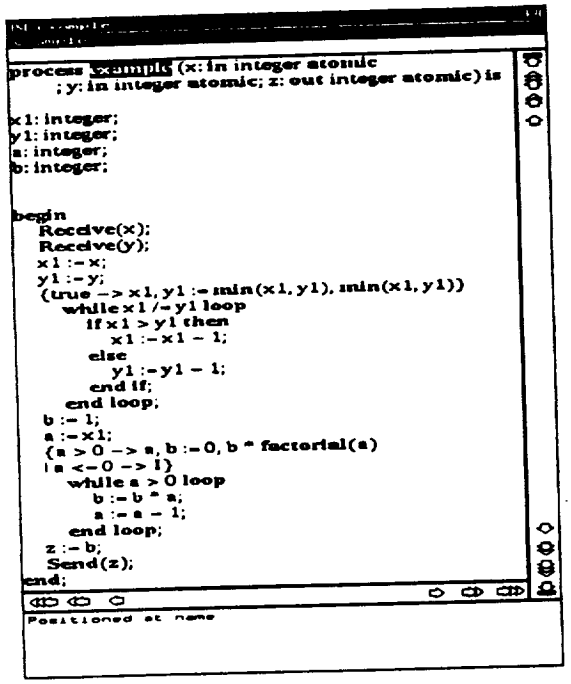

Figure 4: The program to be specified.

4.2.3 **Example:** We describe a short example, due to the space limitation, to demonstrate a sample result obtained using CARE-FSQ$_2$. In order to find the correct specification of a while loop, the user should annotate it with a trial loop function enclosed between two curly braces. CARE-FSQ$_2$ assists the user in verifying the correctness of the loop specification by calculating the composition $[if\ B\ then\ S\ fi] \circ f$. The user, on the other hand, must ensure that the three while loop verification conditions are satisfied. After verifying all the while loops in the program, the user

```
expr : (x1-y1 < 0 or y1-x1 < 0) and y1-x1 < 0
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: y1 < x1

expr : (x1-y1 < 0 or y1-x1 < 0) and not y1-x1 < 0
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: y1 > x1

expr : not (x1-y1 < 0 or y1-x1 < 0)
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: y1 = x1

The symbolic execution result is :
==================================

 y1 < x1 ->
    x1, y1 :=
    min(x1-1,y1), min(x1-1,y1)

 |

 y1 > x1 ->
    x1, y1 :=
    min(x1,y1-1), min(x1,y1-1)

 |

 y1 = x1 ->
    x1, y1 :=
    min(x1,y1), min(x1,y1)
```

Figure 5: Finding the specification of the first loop.

```
expr : -a < 0 and -a+1 < 0
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: a > 1

expr : -a < 0 and a-1 <= 0
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: a = 1

expr : not -a < 0 and a <= 0
Would you like to simplify this expression? [y/n]: y
Enter the simplified expression: a <= 0

The symbolic execution result is :
==================================

 a > 1 ->
    a, b :=
    0, b*GAMMA(a+1)

 |

 a = 1 ->
    a, b :=
    a-1, b*a

 |

 a <= 0 ->
    a, b :=
    a, b
```

Figure 6: Finding the specification of the second loop.

can proceed to find the functional meaning of the whole program.

Figure 4 shows a program that receives two integers as input , finds their minimum, calculates its factorial

if it is positive, and then saves the result in z. First, the verification conditions of the two while loop have to be checked. Hence, we let CARE-FSQ$_2$ print the composition $[if\ B\ then\ S\ fi] \circ f$ to assist us in this process. Before printing the results of the composition, the user is prompted to enter his simplifications for some expressions if he/she desires(see figures 5 and 6).

Since the three verification conditions are satisfied for both loops, we can therefore proceed to find the functional meaning of the whole program which is shown in figure 7.

```
The symbolic execution result is :
==================================

 -min(x,y) < 0 ->
    x, y, z, x1, y1, a, b :=
    x, y, GAMMA(min(x,y)+1), min(x,y),
    min(x,y), 0, GAMMA(min(x,y)+1)

 |

 min(x,y) <= 0 ->
    x, y, z, x1, y1, a, b :=
    x, y, 1, min(x,y), min(x,y), min(x,y), 1
```

Figure 7: Specification of the whole program.

5 Conclusion

In this paper, we have presented a process model for extracting reusable components. It first identifies these components using software metrics, then it qualifies them. We have focused on the qualification phase which generates their formal specifications, generates a significant set of test cases, and packages them for future reuse. We have then described the specification tool of the qualification phase, CARE-FSQ$_2$, that helps in understanding programs by generating their correct formal specifications. Further research needs to be done in order to be able to qualify and tailor large programs for reuse.

Acknowledgement

Research for this study was supported in part by NASA (Grant NSG-5123), ONR (Grant NOOO14-87-k-0307), and Italsiel S.p.A. (IAP Grant).

References

[1] 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.

[2] V. R. Basili, "Software Development: A Paradigm for the Future", *Proc. Compsac'89*, IEEE Computer Soc. Press, Los Alamitos, Calif., Order No. 1964, pp. 471-485.

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

[4] J. D. Gannon, R. B. Hamlet and H. D. Mills, "Theory of Modules", *IEEE Trans. on Software Engineering*, vol. SE-13, no. 7, July 1987, pp. 820-829.

[5] V. R. Basili, G. Caldiera, G. Cantone, "A Reference Architecture for the Component Factory", *Technical Report CS-TR-2607*, Institute for Advanced Computer Studies and Dept. of Computer Science, Univ. of Maryland, College Park, MD 20742, March 1991.

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

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

[8] E. W. Dijkstra, "A Discipline of Programming", Prentice Hall, 1976.

[9] H. D. Mills, V. R. Basili, J. D. Gannon, and R. G. Hamlet, "Principles of Computer Programming: A Mathematical Approach", Boston, MA, Allyn and Bacon, 1987.

[10] T. W. Reps and T. Teitelbaum, The Synthesizer Generator Reference Manual, Springer-Verlag, 1989.

[11] B. W. Char et al, Maple User's Guide, Watcom Publication Limited, Waterloo, Ontario, 1985.

[12] B. Joo, "Adaptation and Composition of Program Components", Ph.D. Dissertation, Dept. of Computer Science, Univ. of Maryland, College Park, Maryland, 1990.