

AD-A170 032

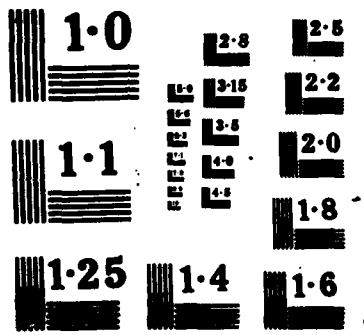
METAPROGRAMMING: A NEW METHODOLOGY FOR THE CONSTRUCTION  
OF QUALITY SOFTWARE. (U) UNIVERSITY OF SOUTHERN  
CALIFORNIA LOS ANGELES DEPT OF COMPUTE. L FLON ET AL.  
30 JAN 86 53-4510-1741 AFOSR-TR-86-0419 F/G 9/2

1/1

UNCLASSIFIED

NL





AD-A170 032

REPORT DOCUMENTATION PAGE

1. REPORT SECURITY CLASSIFICATION <b>UNCLASSIFIED</b>			1b. RESTRICTIVE MARKINGS N/A		
2a. SECURITY CLASSIFICATION AUTHORITY N/A			3. DISTRIBUTION/AVAILABILITY OF REPORT Unlimited distribution		
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A					
4. PERFORMING ORGANIZATION REPORT NUMBER(S) 53-4510-1741			5. MONITORING ORGANIZATION REPORT NUMBER(S) <b>AFOSR-TR. 86-0419</b>		
6a. NAME OF PERFORMING ORGANIZATION University of Southern California		6b. OFFICE SYMBOL (If applicable)	7a. NAME OF MONITORING ORGANIZATION Air Force Office of Scientific Research		
6c. ADDRESS (City, State and ZIP Code) University Park Los Angeles, California 90089-1147			7b. ADDRESS (City, State and ZIP Code) Computer Science Department University of Southern California SAE 200, Los Angeles, Ca. 90089-0782		
8a. NAME OF FUNDING/SPONSORING ORGANIZATION AFOSR		8b. OFFICE SYMBOL (If applicable)	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER AFOSR-81-0199		
8c. ADDRESS (City, State and ZIP Code) Bolling AirForce Base, Bldg. 410 Washington, D.C. 20332			10. SOURCE OF FUNDING NOS.		
			PROGRAM ELEMENT NO. 61102F	PROJECT NO. 2304	TASK NO. A2
11. TITLE (Include Security Classification) <b>Metaprogramming: A New Methodology for the Construction of Quality Software</b>					
12. PERSONAL AUTHOR(S) L. Flon/L.W. Coopriider/E. Horowitz					
13a. TYPE OF REPORT Final		13b. TIME COVERED FROM 6/15/81 TO 9/14/84		14. DATE OF REPORT (Yr., Mo., Day) 1/30/86	
15. PAGE COUNT 11					
16. SUPPLEMENTARY NOTATION					
17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary) (Include block number)		
FIELD	GROUP	SUB GR.			
19. ABSTRACT (Continue on reverse if necessary and identify by block number) There were three major contributions that came out of this research. The first was the development of a program development environment that permits software to be reused. The second was the development of techniques for the design and specification of concurrent programs. The third was a new method for writing programs that involves pictures. For each of these contributions a student Ph.D. thesis was produced, in particular Dr. Anne Curran worked on the first problem, Dr. Thierry Paradan worked on the second problem and Dr. Georg Raeder worked on the last problem. Since each of their contributions are radically different, this summary report is broken into three categories, each based upon their work.					
20. DISTRIBUTION/AVAILABILITY OF ABSTRACT UNCLASSIFIED/UNLIMITED <input checked="" type="checkbox"/> SAME AS RPT. <input type="checkbox"/> DTIC USERS <input type="checkbox"/>			21. ABSTRACT SECURITY CLASSIFICATION		
22a. NAME OF RESPONSIBLE INDIVIDUAL Ellis Horowitz <i>Ellis Horowitz</i>		22b. TELEPHONE NUMBER (Include Area Code) (213)743-6453		22c. OFFICE SYMBOL nm	

DTIC FILE COPY

DTIC ELECTED  
JUL 25 1986  
S D

Title : Metaprogramming: A New Methodology for the Construction of Quality Software

Principal Investigators : Lawrence Flon/Lee Cooperider/Ellis Horowitz

Inclusive Dates : 6/15/81 to 9/14/84

Costs : \$330,922

Junior Research Personnel : Anne Curran, Thierry Paradan, and Georg Raeder

#### ABSTRACT

There were three major contributions that came out of this research. The first was the development of a program development environment that permits software to be reused. The second was the development of techniques for the design and specification of concurrent programs. The third was a new method for writing programs that involves pictures. For each of these contributions a student Ph.D. thesis was produced, in particular Dr. Anne Curran worked on the first problem, Dr. Thierry Paradan worked on the second problem and Dr. Georg Raeder worked on the last problem. Since each of their contributions are radically different, this summary report is broken into three categories, each based upon their work.

Approved for public release ;  
distribution unlimited.

86 7 23 157

## On the Design and Implementation of an Environment for Reusable Software - Anne Marie Curran

As the cost of hardware decreases, software costs increasingly dominate a computer system, particularly when large scale software systems are involved. Why is it that experience gained over the past few decades has resulted in drastic decreases in hardware cost and production time, and in increased reliability, but less dramatic advances have been made in software construction? Obviously, some of the gains in hardware are the result of more sophisticated basic materials, but more sophisticated "materials" have also been created to aid software developers, for example, high level languages.

An examination of the various approaches used in building large hardware and software systems reveals the answer to this question. Hardware construction generally uses the technique of building systems from pre-existing, correct modules, rather than from scratch. This technique, which is also used by other engineering disciplines, is possible only because the components are designed to be reusable and not for a specific application. Just as the nuts and bolts which hold the cabinet of a computer together were not designed and manufactured solely for that purpose, most of the chips in the machine were not designed for that one application. There are occasionally exceptions, of course, where designing a special-purpose component from scratch is necessary. But these situations are time-consuming and costly, and should be avoided whenever possible.

The benefits of building a system from existing components are well-known: lowered costs, increased reliability, and simplified maintenance and repairs. Constructing a large system is simply a matter of connecting the appropriate components

THESE RESEARCH REPORTS ARE AVAILABLE FROM THE  
OFFICE OF SCIENTIFIC RESEARCH (ASPC)  
DEFENSE DEPARTMENT  
WASHINGTON, D.C. 20301  
A report has been prepared for this  
purpose and is available for sale at a special price.  
The price is unlimited.  
THOMAS J. KERPER  
Chief, Technical Information Division



tends to immediately start writing code, in contrast to the hardware designer who will probably pick up a catalog and determine which parts are already available. But the programmer is not entirely to blame, since the traditional programming environments do not support either developing or utilizing reusable software. Suppose that a programmer realizes that someone must have already solved a problem, and wishes to avoid duplicating that effort. The programmer is likely to encounter several problems:

- The programmer must find out that the software exists.
- The programmer must determine how to use it to solve his/her particular problem.
- The software may be similar to, though not quite what the programmer needs.
- The software may be too inefficient.
- The software may not work.

Typically, a software module which is reused has not been designed that way, but consists of some useful part of a system which has been extracted and made available to other programmers. Unfortunately, this approach to generating reusable modules often does not work well. One of the most serious problems is that a module which is designed for a particular application frequently contains implicit assumptions about its environment. If an unsuspecting programmer removes the module from the original environment, these assumptions may be violated, causing the module to work poorly or not at all. Thus an attempt to take advantage of another programmer's work often becomes a frustrating waste of effort.

To be successfully reused, a module must be designed and constructed with that intention. A reusable module is defined here to have the following properties:

- It must be general enough to do precisely what a user needs.

- It must be efficient.
- It must be easily maintained.
- It must be correct.

The qualities of efficiency, correctness and solving the right problem are vital from the user's point of view. If a module lacks any of these properties, the user may be better off writing the code from scratch.

Throughout the discussion of metaprogramming, the term "reusable software" and "general purpose software" are used almost interchangeably. The reason for this is that a special purpose program may have such limited functionality or make so many environmental assumptions that in most cases, it cannot be reused without extensive modifications. The general purpose program has several advantages over one written for a specific purpose. By sharing common code, the general purpose program requires fewer lines of code than the many specialized programs needed to solve a set of problems. Maintenance costs are lower, since someone may neglect to change some of the specialized code, so sharing code also has an advantage in this area.

In practice, however, writing software with all these properties is not an easy task. One of the basic problems is that constructing a general solution to a problem is inherently more difficult than solving one specific instance of that problem. This situation is not unique to the software engineering field, where identifying and then understanding the general problem are often major obstacles. Part of the answer lies in finding programmers who are capable of solving abstract problems and convincing them of doing so. For this to be practical, programmers must be convinced that the extra effort is justified, and they must be provided with a set of tools which make writing and



maintaining high quality, general purpose software as easy as possible. Unfortunately, the tools which are currently available make the task even more difficult. The programmer may often find that the only way to create a module possessing some of the above qualities is by deliberately omitting the others.

An important part of improving software tools is to identify the specific problems that programmers encounter when trying to write or access general purpose software. It seems obvious from the areas already mentioned that a language which is suitable for writing general purpose software is vital, and that accessing these modules requires some sort of a library mechanism. In fact, writing, maintaining, and accessing reusable software does require a complete environment which is appropriate for these tasks. Various aspects of programming environments have been studied, but in general they do not overcome all of the problems.

### 1.2 The Metaprogramming Environment

To solve the problems of writing reusable software, a new methodology, called metaprogramming, is presented in this dissertation. A metaprogram is the simultaneous denotation of a group of closely related programs, called a program family. The metaprogram cannot be directly executed, but rather serves as a template of a solution to a problem. To solve a particular problem, the metaprogram is provided with information concerning the application, thereby instantiating it into an ordinary module. There are several advantages to constructing software using this approach. By encoding an entire family rather than just one program, the generality which is lacking in most library modules is achieved. In addition, the simultaneous representation of the program family permits common code to be shared, which reduces the cost of maintaining the module. The fact that the metaprogram is not in itself executable avoids the

environmental assumptions that tend to creep into ordinary modules.

An important metaprogramming concept is that developing software to solve abstract problems is not only more difficult, but also involves different activities than coding the answer to a given problem. If a programmer is given the same set of tools to solve both classes of problems, the task becomes extremely difficult, if not impossible. However, the actions required to implement both classes are not completely disjoint. In other engineering fields, not only are systems designed from components, but the more complex components are themselves built from simpler components. A similar design principle is also expected when constructing software, so that larger, more complex software modules are built by interfacing smaller modules. Thus the metaprogrammer is supplied with a superset of the tools given to the ordinary programmer in order to also take advantage of previously written modules.

A high-level language that is suitable for expressing general purpose programs is an integral part of the metaprogramming system. Unfortunately, conventional programming languages are not powerful enough to encode non-trivial program families. Since they are not designed to solve abstract problems, conventional languages are incapable of expressing the variations in a program family so that its instances are both reliable and efficient. Usually the only alternative is to choose between making the program reliable and making it efficient. If reliability takes precedence, the programmer must resort to using mechanisms which are expensive at runtime, such as variant records, and encode all the variations to allow semantic checking. Overhead is incurred from the program determining what problem to solve, as well as from storage which is wasted by routines and data structures which are not needed by an application. If efficiency has priority, then the only mechanisms for encoding variations are forms of

source-level text manipulation such as conditional compilation or macros. These may be efficient in that the unneeded parts of the program will disappear during compilation, but only those variations which are actually used are known to be correct. In addition, there are variations in program families which are impossible to express in a general fashion, even if reusability and efficiency are ignored

For these reasons, a special high-level language for writing metaprograms was developed by augmenting the DOD language Ada. The major strength of this language, called meta-Ada, is that it allows simultaneous semantic checking of the entire program family during compilation, yet only those parts of the program which are actually needed for a specific member exist at runtime. The complete metaprogramming environment also requires library and runtime support. Modules interface information is an important part of the library, just as it is in more conventional environments. In addition, the library must contain functional descriptions of the module in a format accessible either by users or by system programs, and verification information for the code in each module. The module description also provides guidance in choosing a legal and efficient instance of the unit according to how it is referenced by an application.

### 1.3 Goals

The purpose of this research is to find feasible solutions to some of the problems of writing reliable, general purpose software. This is accomplished by the following:

- Identification of the problems which exist in the traditional software development environment in both developing and using general purpose software, with emphasis on those problems which are caused by conventional programming languages.
- Development of constructs which remedy the inadequacies in programming languages.
- Integration of these constructs into a suitable high-level programming

language.

- Identification of the issues involved in implementing a metaprogramming language.
- Demonstrating that the implementation is both possible and feasible by building a prototype compiler and linker for this language.

The latter goal is particularly important, as we are not interested in a highly theoretical discussion of abstract programming but research that can be applied to actual software development. Our goal is not only to demonstrate that the proposed language can solve particular problems in writing general purpose software, but that the implementation of metaprogramming constructs is both possible and feasible. The term "feasible" is used here to mean that the time and storage requirements for supporting metaprogramming are not prohibitive. Since the prototype system was not designed as a production system, efficiency was not the major concern. However, if the prototype appears feasible, then its cost can form a crude upper bound for a production system.

In translating a metaprogramming language, a number of issues arise which are not normally encountered when translating conventional languages. Since the metaprogram represents an entire program family, semantic checking of all its variations must be performed. This introduces problems such as keeping track of which parts belong to the different instances, and developing algorithms for checking constructs which may not always exist.

Although an important part of a production software system, the design and implementation of the metaprogramming library and runtime environment are largely beyond the scope of this research. A simple library mechanism has been implemented in the prototype system, and library and runtime support are discussed in terms of

language design and implementation decision.

### **A Unified Approach to the Construction of Correct Concurrent Programs - Thierry Paradan**

Although a great deal of attention has been devoted to the problem of concurrent program verification, a unified framework for constructing shared memory and distributed programs and facilitating their verification has remained largely unexplored. The purpose of this thesis is to explore an approach to this unification.

To achieve this goal, a simple programming language (called *deb*) based on a construct related to Dijkstra's "do..od" is proposed. One of the fundamental features of *deb* is the absence of a classical synchronization primitive. Instead, to ensure harmonious cooperation among concurrent processes, the concept of "well-formedness" is introduced. What characterizes a well-formed concurrent program is its semantic equivalence to a sequential nondeterministic program. To verify that a program is well-formed, a sound proof system is supplied in the form of a set of conditions that guarantee well-formedness. The main advantage of this set of rules is a greater clarity in the conceptual understanding of the problems arising from the interaction of concurrent agents.

Since a well-formed program is semantically equivalent to sequential nondeterministic one, verification methods defined for sequential programs can readily be applied. Further, it is argued that the design of *deb* structures can serve as a basis for the definition of a large set of powerful proof rules. The expected consequence is a significant decrease, both in length and difficulty, of program verification in this language.

In addition to accomplishing its primary objective, *deb* has the following desirable

features.

1. While it is tailored for use in distributed programming, deb allows the use of shared data and does not contain any specific message passing primitive. Messages are defined through a special type of shared data object called a "switch". Programs written in deb can thus be viewed and used as ordinary (shared memory) parallel programs.
2. In addition, the concept of well-formedness facilitates the implementation of sequential nondeterminism by making it easier and more efficient than in the general case.

### **Programming in Pictures - Georg Raeder**

Programming in conventional programming languages is awkward because the resulting programs are quite far from how we like to think about them, both conceptually and representationally. By representing programs as semantically suggestive graphical images we can shorten the gap between mind and medium and thereby make programming more pleasant, efficient and less error-prone. A rich graphical interface can also aid naive programmers by making abstract concepts concrete.

In this thesis we examine the use of pictures in programming. We point out the salient characteristics of pictures vs. text. In particular, the concreteness, random access nature, high transfer rate, namelessness, multi-dimensionality, and possibilities for animation render pictures well qualified for representing programs. We discuss in more detail the best form of a pictorial program display, and arrive at a solution based on data structures as the primary displayed aspect. We also assign other program aspects, like control and hierarchy, to picture dimensions, obtaining a unified view that allows the representation of programs as a single object instead of a series of different views. We develop techniques for reading and writing programs through sequences of pointing actions animated on top of the data structure display, mimicking the way people

informally explain programs through handwaving on data structure illustrations.

We describe a concrete implementation of our ideas about programming in pictures. The system is based on a functional programming model. It allows the creation of functions where the data types of input and output objects are illustrated by the user. The user inputs pictures related to the application domain and inserts them in an algorithm framework supplied by the system. We show how our style of programming leads us to a version of programming by example.

Finally, we examine what audience would benefit the most from a pictorial programming system, and what kind of applications these people would be interested in. We define the term "casual programming" as the creation of small programs by naive or casual users, and identify this as a useful application area of programming in pictures.

END

DTIC

9-86