Working Paper 188

May 1979

Evolutionary Programming with the Aid of A Programmers' Apprentice

Carl Hewitt

A.I. Laboratory Working Papers are produced for internal circulation, and may contain information that is, for example, too preliminary or too detailed for formal publication. Although some will be given a limited external distribution, it is not intended that they should be considered papers to which reference can be made in the literature.

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for this research was provided in part by the Office of Naval Research of the Department of Defense under Contract N00014-75-C-0522.

Artificial Intelligence Laboratory

Massachusetts Institute of Technology

© MASSACHUSETTS INSTITUTE OF TECHNOLOGY 1979

Evolutionary Programming with the Aid of a Programmers' Apprentice

Carl Hewitt

M.I.T.

Room 813
545 Technology Square
Cambridge, Mass. 02139
(617) 253-5873

I -- EVOLUTIONARY PROGRAMMING

The documentation, implementations (we use the plural because we want to allow for multiple implementations), and runtime environment of useful software systems evolve asynchronously and continually. This is particularly true of large systems for applications such as reservations, programming environments, real-time control, data base query and update, and document preparation. Implementations change because of the development of new hardware and algorithms. Documentation (including tutoring programs such as [Burton and Brown: 1976, Goldstein: 1976, Genesereth: 1979, and Miller: 1979]) changes to keep up with other changes. Runtime environments change because of changes in legislation and other unforseen events rearrange the physical environment.

Neither fully automatic program synthesis nor fully automatic program proving have been very successful so far in dealing with large software systems. We believe that it is necessary to build environments to interact with software engineers in the course of the **co-evolution** of the partial interface specifications and implementations of a system. Realistic software systems impose the requirement that the interface specifications of modules must be allowed to evolve along with the implementations. This situation makes it correspondingly more difficult to construct a fully automatic programmer for such systems. In case of inconsistency between the partial interface specifications, runtime environment, and implementation of a large system, it may be desirable to modify *any* of them. It is naive to believe that complete interface specifications can be laid down once and for all time in a large software system and the implementations of the modules derived by top-down stepwise refinement.

It is important to realize that the co-evolution of implementations and interface specifications is an entirely natural and fruitful process. In most applications it is fruitless to delay implementation until complete and final interface specifications have been provided.

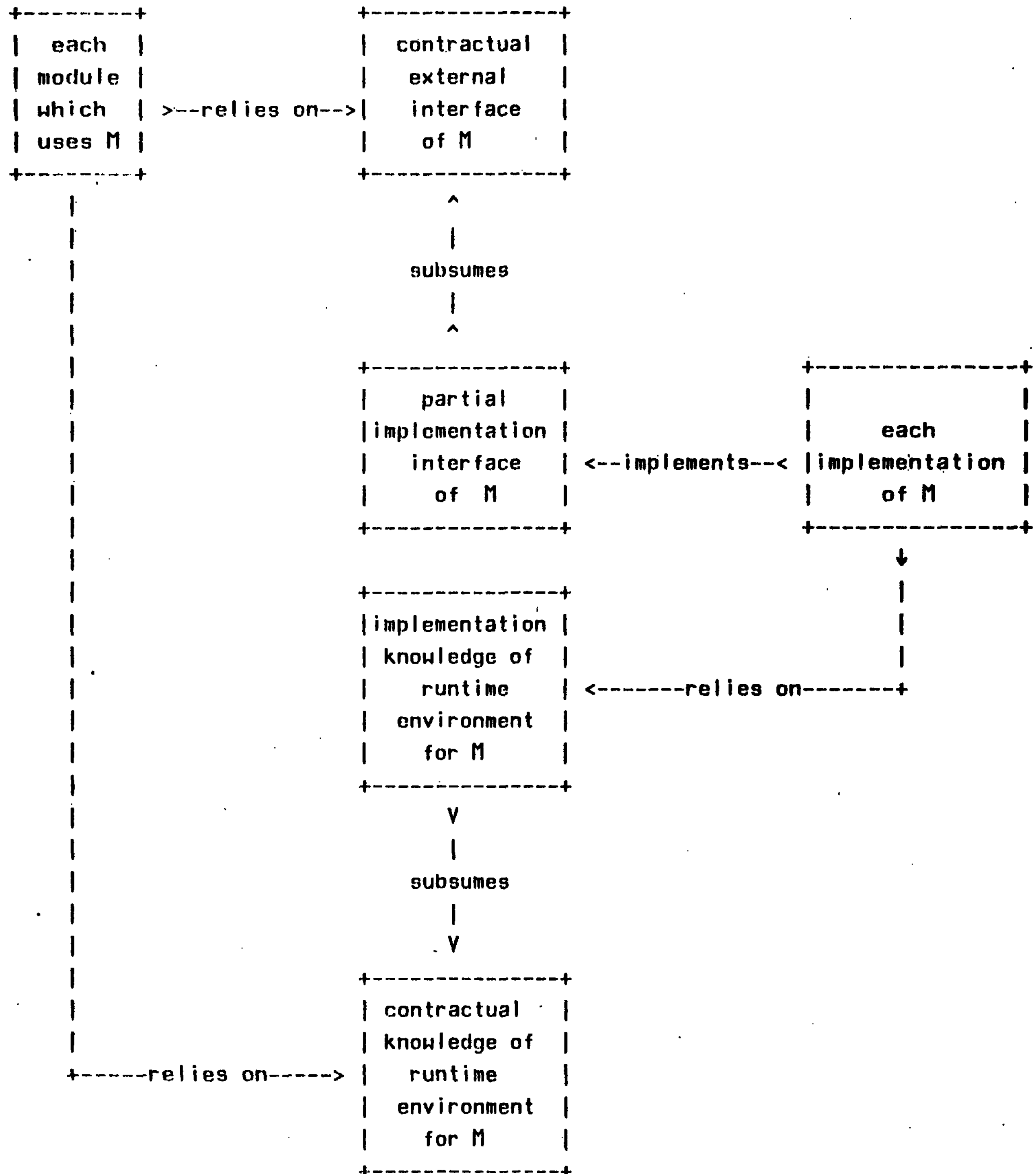
The history of the development of text editors on interactive systems provides a good illustration of the co-evolution of implementations and interface specifications. In the late fifties when text editors were first being developed, it would have been completely impossible to have developed interface specifications or implementations for current generation text editors. It was necessary for users and implementors of text editors to evolve the systems over a long time period in the context of an evolving hardware base in order to reach the current state of development. Furthermore, it seems rather clear that interactive text editors will continue to evolve at a rapid pace for quite some time in the future.

Exploration of what it is possible to implement provides guidance on what are reasonable partial interface specifications. As experience accumulates in using an implementation, more of the real needs and possible benefits are discovered causing the partial interface specifications to change. An important consideration in a proposed change is the difficulty of modifying the implementation and documentation. Conversly, implementors attempt to create systems that have the generality to cope with anticipated directions of evolution. Partial interface specifications in large systems change gradually over a long period of time through a process of negotiation.

II -- A PROGRAMMERS' APPRENTICE

A group at M.I.T. is engaged in a long term research effort to build an interactive system [called the Programmers' Apprentice] to aid in the construction and evolution of large software systems using partial, multiple, incremental interfaces between users and implementors of software systems. The Programmers' Apprentice effor [Rich, Shrobe, Waters, Sussman, and Hewitt: 1978; Shrobe: 1978; Waters: 1978; Hewitt: 1978; and Rich, Shrobe, and Waters: 1979] builds on antecedent and similar work by [Floyd: 1971; Hewitt: 1971; Sussman: 1975; Hewitt and Smith: 1975; Rich and Shrobe: 1976; Yonezawa: 1977; and Moriconi: 1978].

The following diagram shows the relationship between the users of a module M, its partial multiple interface specifications, its implementations, and the knowledge of the runtime environment of M.



The contractual external interface of a module M should be as close as possible to an absolute interface in the sense that any external module which uses M should only rely on properties of M implied by its external interface and the contractual knowledge of the runtime environment for M. Notice that associated with each implementation of M, we have versions of the interface specification and knowledge of the runtime environment that are private to each implementation. The private versions contain the documentation that is special to each one.

The contractual knowledge of runtime environment is the shared knowledge relied on by both the users and implementors of M. Examples are the laws of physics in a bubble chamber analysis program, the tax law for an income tax preparation program, models of the behavior of jets and radars in an air traffic control program, and the number of physical tracks on a disk for a memory management module.

A primary goal of the Programmers' Apprentice is to make explicit how each module depends on the partial interface specifications of other modules and the knowledge of the runtime envirnment, how each implementation of a module meets its partial interface specifications, and how each implementation depends on the knowledge of the runtime environment. The proposed Programmers' Apprentice will gradually make the above dependencies explicit through a process of **symbolic evaluation** [Deutsch: 1973; Hewitt and Smith: 1975; Yonezawa: 1977; King: 1976; Clarke: 1976; Shrobe: 1978; Hewitt: 1978; and Cheatham, Holloway, and Townley: 1978]. Symbolic evaluation consists of executing the implementation of a module M on abstract input using the partial interface specifications of modules it uses. An important purpose of symbolic evaluation is to make explicit exactly how the partial interface specifications of M are satisfied. Symbolic evaluation ensures that a module M only depends on the partial interface specifications and the knowledge of the runtime environment of the modules which it uses and does not depend on idiosyncratic properties of particular implementations. It establishes and maintains an interface between users and implementors of a module. An explicit record of dependencies is necessary for the successful creation and co-evolution of the documentation, implementations, and knowledge of runtime environment of a large software system.

Evolving systems of the kind we are describing will require the capabilities of expert programmers for a long time into the future. Our proposed Programmers' Apprentice plays mainly an advisory and bookkeeping role. We believe that this state of affairs is entirely appropriate given the current state of the art in fully automatic program synthesis and program proving.

III -- A DESCRIPTION SYSTEM

One fundamental tool in our approach is a description system (being developed jointly with Giuseppe Attardi) which can be used to describe properties of modules to the Programmers' Apprentice. It is intended to facilitate use of the following kinds of descriptions:

> **PARTIAL descriptions** are used to express whatever properties of system. Descriptions of realistic systems such as air traffic control involve ineviatable simplifications and approximations. It is useless to wait for a complete description of an air traffic control system because the goal of complete description is unattainable.
>
> **INCREMENTAL descriptions** which enable us to further describe objects when more information becomes available and are a necessary feature for the effective use of **partial descriptions.** For example at some point as the velocity of jet airplanes increases it will be necessary to take the Coriolis effect into account in the air traffic control system.
>
> **MULTIPLE descriptions** which enable us to ascribe multiple overlapping descriptions to an object which is used for multiple purposes. Multiple descriptions are important in multiple specifications and proofs because different properties of an object might be useful in different contexts.

Our description system is used in stating partial specifications of programs, as a powerful flexible notation to state type declarations, and as a notation to express conditions that are tested during program execution. The assumptions and the constraints on the objects manipulated by a program are an integral part of the program and can be used both as checks when the program is running and as useful information which can be exploited by other systems which examine the program, such as translators, optimizers, indexers, etc. We believe that bugs occurring in programs are frequently caused by the violation of implicit assumptions about the environment in which the program is intended to operate. Therefore many advantages can be drawn by a language that encourages the programmer to state such assumptions explicitly and by a system which is able to detect when they are violated.

IV -- ACKNOWLEDGEMENTS

The development of the ideas expressed in this paper has been the work of a large group of people over many years. The intellectual roots go back to the early emphasis by Minsky and Papert of the importance of "debugging" in problem solving (especially programming). I am indebted to my colleague Gerry Sussman for many an inspiring discussion on how to best incorporate these ideas in practical systems. In turn we are indebted to our thesis students Jerry Barber, Chuck Rich, Howie Shrobe, Dick Waters, and Aki Yonezawa for providing the hard work and ideas necessary to make this area into more of a science. The current members of the Programmers' Apprentice Research Group include Beppe Attardi, Jerry Barber, Carl Hewitt, Henry Lieberman, Chuck Rich, Howie Shrobe, Maria Simi, Gerry Sussman, and Dick Waters.

Michael Genesereth and Chuck Rich made some valuable suggestions which materially improved the presentation of this paper.

V -- REFERENCES

Burton, R. and Brown J. S. "A Tutoring and Student Modeling Paradigm for Gaming Environments" SIGCSE Bulletin. Vol. 8 No. 1. February 1976. pp. 236-246.

Cheatham, T. E.; Holloway, G. H.; and Townley, J. A. "Symbolic Evaluation and the Analysis of Programs" Aiken Computation Laboratory. Harvard University. TR-19-78. November 1978.

Clarke, L. "A System to Generate Test Data and Symbolically Execute Programs" IEEE TSE-2 No. 3. Sept. 1976. pp 215-222.

Deutsch, P. "An Interactive Program Verifier" Report No. CSL-73-1. Xerox PARC. May 1973.

Genesereth, M. R. "The Role of Plans in Automated Consultation" IJCAI-79. Tokyo, Japan. August 1979.

Goldstein, Ira P. "The Computer as Coach: An Athletic Paradigm for Intellectual Education" MIT AI Memo 389. December 1976.

Hewitt, C. and Smith, B. "Towards a Programming Apprentice" IEEE Transactions on Software Engineering. SE-1, #1. March 1975. pp. 26-45.

Hewitt, C.; Attardi, G.; and Lieberman, H. "Specifying and Proving Properties of

Guardians for Distributed Systems" MIT AI Lab Working Paper 172. December 1978. Revised April 1979. Proceedings of International Symposium on the Semantics of Concurrent Computation. Evian-les-bains, France. July 1979.

Hewitt, C. "Evolving Parallel Programs" MIT AI Laboratory Working Paper 164. December, 1978. Revised April 1979.

King, J. C. "A New Approach to Program Testing" IBM Research Report RC-5037. September 1974.

Miller, M. L. "Planning and Debugging in Elementary Programming" Unpublished Doctoral Dissertation. MIT. February 1979.

Moriconi, Mark S. "A Designer/Verifier's Assistant" SRI Technical Report CSL-80. October, 1978.

Rich, C. and Shrobe, H. "Initial Report on a LISP Programmer's Apprentice" December 1976. AI-TR-354. IEEE Transactions on Software Engineering. SE-4. No. 6. November 1978. pp. 456-467.

Rich, C.; Shrobe, H. E.; Waters, R. C.; Sussman, G. J.; and Hewitt, C. E. "Programming Viewed as an Engineering Activity" MIT A.I. Memo 459. January 1978.

Rich, C.; Shrobe, H. E.; and Waters, R. C. "Computer Aided Evolutionary Design for Software Engineering" MIT A.I. Memo 506. January 1979.

Shrobe, H. "Logic and Reasoning For Complex Program Understanding" MIT PhD. Thesis, October 1978.

Sussman, G. J. "A Computer Model of Skill Acquisition" American Elsevier. 1975.

Waters, R.C. "Automatic Analysis of the Logical Structure of Programs" MIT AI Laboratory TR-492. December 1978.

Wulf, W. A. "Abstraction and Verification in ALPHARD: Introduction to Language and Methodology" ISI/RR-76-46. June 1976.

Yonezawa, A. "Specification and Verification Techniques for Parallel Programs Based on Message Passing Semantics" MIT/LCS/TR-191. December, 1977.