A Preliminary Philosophy for ARCTURUS An Advanced Highly-Integrated Programming Environment

Thomas A. Standish

Irvine
Programming Environment Research Center

Computer Science Department
University of California
Irvine, California 92727

April 23, 1980

Technical Report 150

A b s t r a c t

At Irvine, we are currently in the initial stages of designing a programming environment, called Arcturus. This paper is a report of work in progress giving our preliminary philosophy and expressing preliminary thoughts on an initial Arcturus design.

Arcturus is an advanced, highly-integrated programming environment intended for use in the late 1980s. We assume that programmers will each be equipped with large flat-screen displays driven by powerful desk-top computers linked into local networks by high band-width channels, and that shared central resources such as archival databases and multifont printing systems will be available.

Arcturus is aimed at "programming in the large", that is, programming by many people, on large programs, with maintenance lifetimes of many years. In such a user setting, problems of management, documentation, training, testing, version control, diagnosis, and debugging must be solved effectively by people who, for the most part, are not authors or designers of the original system.

Some preliminary design concepts that Arcturus supports are as follows:

(1) Arcturus supports a "rapid prototyping" language --- a very high level, strongly extensible language useful for rapid construction of working prototypes of systems (emphasizing cheap, rapid construction at the expense of running efficiency and polish).

(2) Arcturus supports refinement of these prototype programs, or protoprograms, for short, into programs written in program design languages (or PDLs), which express designs. PDL programs are ultimately refined into concrete, detailed, optimized programs expressed in an implementation language.

(3) Arcturus supports a computer-based form of program documentation in which program forms at various levels of abstraction can have attribute/value pairs attached to any of their granules (granules being well-formed program units of any size such as constants, variables, operators, expressions, statements, blocks, and modules) and in which the attributes may be selectively viewed and queried to suit the needs of different audiences.

(4) The notion of attribute/value attachment to granules of program forms also supplies the principal mechanism for promoting a high degree of environment integration. By attaching to program granules such

attributes as clocks, counters, units of programmer and system resources spent, version descriptions, access controls, descriptions of tests passed, task schedule data, computer sizing estimates, and so on, smooth integration between the activities of designers, managers, testers, maintainers, programmers, and documenters can be achieved, and environment tools can cooperate with each other conveniently.

(5) Arcturus supports an advanced programmer's workstation, an interactive programmer's notebook, and extensive software management support tools.

In the framework of the Arcturus effort, we have attempted to rethink afresh issues of epistemology related to the programming process that impact documentation, fault diagnosis, maintenance, training, and software upgrade, so that the design of Arcturus will reflect the relationships between the different kinds of expertise that are required in the programming process. We are also attempting to formulate <u>theories</u> of documentation, debugging, and maintenance to guide the development of computer-based support capabilities that assist in the performance of these activities.

In this context, this paper contains preliminary, tentative expositions of background <u>philosophy</u> and <u>rationale</u> that guide our present thinking about Arcturus.

Table of Contents

SECTION Ø

I n t r o d u c t i o n

This paper is a report on preliminary philosophizing in progress at Irvine. We are currently in the process of dreaming up a "far out" programming environment, called Arcturus.

We think the time is ripe to exercise fresh imagination and to dream about novel future programming environments. In the absence of such dreaming, it is quite possible that the new generation of cheap, powerful computers will arrive on our desktops, and we will have only worn out ideas for what to do with them. The dreaming and philosophizing in this paper is intended to portray possible new ways of taking advantage of this equipment, once it becomes available.

We have tried to free ourselves from the restrictive mentality of contemporary time-shared computers. For example, programming in Interlisp can rapidly sink a contemporary PDP-1Ø because of large core loads and high swapping demand. On the other hand, if one has a million word desktop computer with a fifty nanosecond cycle time on one's desktop, Interlisp resides in less than fifteen percent of the memory and not only is the machine ten times faster than a PDP-1Ø, but Interlisp is resident and need not be paged or swapped. Thus, while one can view users of Interlisp as having a "Rolls Royce" mentality on current time-shared machines, perhaps one should view them as having only a "Volkswagen" mentality on the coming generation of desktop machines. We believe, for instance, that in the new generation of desktop machines, it will often be profitable to exchange processing power for ease of expression, as in the "rapid prototyping languages" we introduce and discuss later.

Our dreams are also driven by strongly pragmatic considerations derived from a knowledge of activities in the software lifecycle.

We also feel the time is ripe to rethink deeply the programming epistemology that lies at the root of the programming process. So we have been tackling questions such as the following:

What types of knowledge are used in the construction of application programs? How does application domain knowledge differ from the knowledge used in the concrete implementation domain, in modelling domains, and in collateral reasoning domains?

How should these various types of knowledge be reflected in program documentation, and how can they be used effectively: (a) during training to learn about how a program works, (b) in diagnosis to detect a program error, and (c) during program upgrade when the program is being altered to do something more ambitious?

Is paper a bad medium to use to hold all the different kinds of knowledge representations that are needed in adequate program documentation, and could we do better by using objects of unbounded extent attached to program granules in a database and made selectively visible via computed "views"?

In thinking about Arcturus, we have made the (probably conservative) assumption that in the very near future we will have powerful, cheap, desk-top computers available to each manager and programmer which are linked into networks that provide communication to others and to shared, central resources, such as archival databases, libraries, and peripheral devices. Our dreams for the "Arcturus Personal Workstation" and for "Programmer's Interactive Notebooks" are sketched briefly in what follows.

One of our most important pursuits in the design of Arcturus is to provide a means for <u>rapid</u> <u>prototyping</u> of systems. Requirements analysis is an iterative learning process --- one that can be effectively accelerated by exposure to the behavior of working prototypes which have been built rapidly and cheaply.

For example, in some branches of industry and government, it is not uncommon that three to five years are spent building a system that may be subsequently determined to be non-responsive to user needs, and the system requirements analysis is iterated in succeeding procurement cycles. This pattern is especially evident in systems we are attempting to construct for the first time that are expected to have novel capabilities.

In addition, requirements change rapidly. We may have a working system in the field that we are happy with, and in a matter of hours, the requirements may change (as in the example of the ECM boxes in recent Middle Eastern conflicts).

Thus, rapid prototyping has to do with quick response to changing requirements as well as rapid development in the first place.

Why do we think rapid prototyping systems are a useful pursuit, and why do we think we can be successful implementing them? First, if we relax optimization constraints, we can build models at less construction expense. Second, mock-ups can often yield samples of system behavior adequate to determine responsiveness to user needs at a fraction of the cost of real systems. Third, in building a prototype, often one need not model everything. Instead, one need model only things relevant to the functionality of the system as viewed by the user. Finally, experience with extensible languages provides initial confidence that rapid prototyping systems can be built effectively. We can cite experience demonstrating that we are already successful in this endeavor, and we can build upon these successes to develop a practical, effective capability in our future programming environments.

In a subsequent section, we give a brief view of how we believe a rapid prototyping language can be incorporated in Arcturus.

Arcturus supports _Program Design Languages_, or PDLs, for short. Given an implementation language in which we are to write eventual programs, such as Ada or Pascal, we create a variant of the Arcturus family supporting each such implementation language. This yields, for example, _Arcturus for Ada_ or _Arcturus for Pascal_.

Let L be an implementation language, and consider the PDLs in _Arcturus for L_. Suppose that we take programs in L and we replace actual constants, variables, and operators in these programs with abstract, uninterpreted letters and operator symbols. These uninterpreted symbols then become substitution points for which we can substitute new constants, variables and operators in new semantic domains. We use the term _shell programs_, or _L-Shells_ to refer to the L-programs containing uninterpreted symbols as substitution points.

PDL programs result from taking shell programs and substituting objects and operations in higher level design domains. As a result, the PDL programs have the same shell syntax for their control structures and have the same general appearance as programs in L. Furthermore, they can be processed by many of the tools that process concrete programs in L (such as parsers, pretty-printers, and documentation processors). Additionally, programmers trained to read programs in L can instantly read and write programs in the PDLs for L. Finally, PDL programs at several levels of abstraction can be used in system documentation to relate programs written at different levels of abstraction spanning the gap from the application domain down to the implementation language domain.

A _rapid prototyping language_ can be devised by starting with a variant of PDL, which consists of extending the shell-substituted PDLs by another transformation --- that of replacement of program granules of any size (e.g. constants, variables, operators, procedure calls, expressions, statements, blocks, modules, or any other well-formed program unit) by bracketed descriptions of operations, activities, relationships, and properties that may later be refined into concrete implementations. An example of such a bracketed description is {Compute Incremental Coriolis Force Using Input (C1) and Update

Display(G3)}. The bracketed descriptions are intended to be self-documenting at the design level, and can possess implementations that turn them into self-replacing macros which expand into lower-level PDL text (including, ultimately, text in L).

Attribute/value pairs can be attached to granules of each of these three levels of language: The top-level rapid prototyping language, the intermediate-level PDLs, and the low-level concrete implementation language L. Arbitrary attribute/value attachment provides the basis for tool integration in Arcturus, and is the principal data form used by many of the sophisticated tools.

An important observation to make at this point is that different programming environments are appropriate for different user settings. If one asks the question, "What characterizes a good programming environment?", one tends to be misled into thinking there are absolute answers to this question. In fact, the question itself is a poor question, since whether an environment is good or not should properly be characterized relative to particular circumstances of its use.

We are familiar with the notion that different engineered artifacts are built to suit different purposes. For instance, some airplanes are built to carry 300 people plus baggage on 10,000 mile flights in any kind of weather, and other kinds of airplanes are built to dust crops. We do not expect crop-dusters to be good for carrying 300 people plus baggage on 10,000 miles trips, nor do we expect Boeing 747s to be good at dusting crops cost-effectively. Thus, it is a bad question to ask, "What is a good airplane?"

Likewise with programming environments, some environments are good for programming in the small --- i.e. building programs with short useful lifetimes by small numbers of people with little or no personnel turnover, wherein the maintainers are the same as the implementers and where there is little need for extensive documentation, careful management, training aids, independent validation, resource monitoring, careful version control, and a host of other activities. Yet other environments have been built and extended to deal with programming in the large, where some of the latter activities are of key importance.

We call the characteristics of the user organization in which an environment will be used a "<u>setting</u>". Given the characteristics of a user setting it becomes meaningful to ask whether a given environment design is good for that setting.

So that we will be clear as to the setting for which Arcturus is being designed, we spell out in the next section, the purposes for which Arcturus is intended to be used, and we comment on characteristics of Arcturus that are aimed at dealing with the activities and problems of that setting. This portrayal of the setting and the purposes of Arcturus will set the context for the discussion of mechanisms and features treated in the subsequent sections and will provide the context for discussing the rationale for facets of the design of Arcturus.

SECTION 1

T h e   P u r p o s e   o f   A r c t u r u s

1.1 What is Arcturus?

Arcturus is a programming environment. It is intended to provide support for the activities that take place during the software lifecycle using a computerized medium.

1.2 Software Quality Goals

The goal of Arcturus is to help teams of people produce quality software. Software quality has many dimensions.

1.2.a For instance, there are general dimensions, such as reliability, correctness, efficiency, maintainability, responsiveness to user needs, timeliness of delivery, unit cost, and transferrability.

1.2.b In addition, some software application settings involve special dimensions of software quality , such as quality in parallel processing, fault-tolerance, self-diagnosis, meeting real-time constraints, commercial marketability, and modifiability in the face of rapidly changing system requirements.

1.2.c While different software applications may share the same general software quality dimensions, different applications may have differing sets of special software quality dimensions.

1.2.d In general, when considering software quality goals, no single goal is to be pursued at the expense of the others and no single optimization criterion can be established. There is no point in maximizing one dimension of quality at great expense when so doing reduces some other dimension of quality below a required threshold level.

1.2.e Thus, in Arcturus, software quality goals are viewed as simultaneous constraints to be satisfied rather than as measures to optimize independently of one another. In a sense, then, Arcturus is a <u>constraint</u> <u>satisfaction</u> <u>system</u>.

1.3 <u>Programming</u> <u>in</u> <u>the</u> <u>Large</u>

Arcturus is designed to deal with "programming in the large", that is, programming by large numbers of people, on large programs, with maintenance lifetimes of many years. By contrast, "programming in the small," is programming by a few people, on a small program, with a short useful lifetime.

1.3.a In any large programming project that extends over many years, there is likely to be personnel turnover. This implies that new project personnel will have to read and understand programs written by others. In such a context, training and documentation play key roles in the effectiveness of a programming organization. Arcturus is designed to support effective training and documentation.

1.4 Support of Software Management

According to Barry Boehm [Boehm, 4th Int. Softw. Engr. Conf., Munich 1979], good management disciplines may be the single most important factor in assuring the success of software projects of substantial size. Boehm reports that a major percentage of failures in software projects are attributable to management failures, as opposed to technical failures [Boehm 1979, *op cit.*].

1.4.a *Effective Management Support* Accordingly, Arcturus is oriented toward providing effective management support both to software managers and to individual programmers and programmer teams.

1.4.b *For Programmers* Arcturus has Interactive Programmer's Notebooks to assist programmers in being thorough in carrying out programming chores.

1.4.c *For Managers* Arcturus has earned value reporting systems and tools for resource estimation, measurement, and management by exception. It has interactive letter systems together with reply summary and status reporting systems for managers to use to keep track of project activities, and it has facilities for planning and adjusting resources to fit changing task situations. Also, it has an Interactive Management Interview to assist managers in throughness of project planning, monitoring, and scheduling, and to encourage adherence to software project practices of proven effectiveness.

1.5 High Degree of Integration

Arcturus is a highly integrated programming environment in a number of respects.

1.5.a Program Design Languages are available for use by designers, programmers, and managers.

1.5.a.1 Managers can use designs written in PDLs to do project cost estimation, computer sizing estimates, critical path scheduling, project performance monitoring, and design of task schedules for independent validation, testing, and integration.

1.5.a.2 Programmers can use PDL representations to drive implementation schedules and to organize the structure of their personal programming tasks using their Interactive Programmer's Notebooks.

1.5.a.3 Designers can design in PDL and can follow mechanically derived design walkthrough schedules to catch errors early in the software lifecycle.

1.5.a.4 Maintainers can use PDL representations to get an idea of what modules do at concise, lucid, abstract levels of description.

1.5.a.5 Documenters can use PDLs to assist in the description of what actual implemented modules do.

1.5.a.6 New Programmers undergoing training can read PDL programs to develop an understanding of the modules they will be working on.

1.5.b Integration of Tools and User Interfaces Another form of integration in Arcturus is in the user interface conventions.

1.5.b.1 Certain user functions such as the Help System, the Mail System, and the Calendar and Reminder System are pervasive and can be called in a nested fashion anytime, during any activity. The interrupted activity sits in a background mode during the interruption of the intervening pervasive foreground activity, and the interrupted activity can be resumed after the pervasive activity is terminated.

1.5.b.2 Arcturus Tools are designed to be nestable in this fashion and to permit a range of granularities of interaction.

1.5.b.3 The Arcturus tool set is thus highly integrated, and follows careful integration conventions.

1.5.b.4 The user interface is designed to be simple, pervasive, and highly integrated.

1.5.b.5 Prefabricated user-interface extension procedures for extending On-Line Manuals, Help Systems, Error Reports, Command Completion, Command Syntax, Windowing, and Menuing are available to programmers who wish to add new tools to an Arcturus Environment.

1.5.b.6 Thus, Arcturus supports uniform extension of its command interface language and it encourages users to add new tools that are consistent with the Arcturus tool interface conventions. These conventions ensure that tools obey common, simple, uniform interface rules. Incentives are provided in the form of ease of extension and prefabricated tool-building packages.

1.6 Rapid Prototyping

Arcturus contains a rapid prototyping language enabling rapid, cheap construction of working system prototypes. The rapid prototyping language is strongly extensible and operates at a very high level. Exposure to the behavior of programs written in it is intended to help users learn whether meeting the stated requirements will satisfy true user needs, and to help learn whether the stated requirements are complete, accurate, and fully articulated. User exposure to working prototypes is intended to accelerate the learning process involved in finding out whether the requirements are adequately stated, leading to improved accuracy of the requirements statements at early stages in the system lifecycle and to consequent elimination of wasted effort downstream.

1.7 Arcturus Designed for Future Hardware

Arcturus is designed in anticipation of a coming generation of hardware. Conservative estimates allow us to predict that desktop computers a few years in the future will have memories of on the order of a million words and sub-microsecond/32-bit wide central processors for on the order of a few thousand dollars.

1.7.1 Personal Workstation

Arcturus is designed to take advantage of a mode of computing in which each programmer and manager is equipped with a personal workstation of this sort which is linked into a network that accesses a central database, and shared high-speed, hard-copy devices, and which connects to the outside world over external computer networks if desired.

1.8 Arcturus Emphasizes Maintenance and Upgrade

Maintenance and upgrade are major cost elements in long-lived software systems. Arcturus is oriented toward providing effective support of maintenance and upgrade activities through careful version control, tools for control of releases of families of programs, and automation techniques for handling trouble reports, pending errors and desired upgrades, and enforcement of current documentation practices.

1.9 Validation of Arcturus

A goal of Arcturus R&D is to find a realistic testbed in which it may be convincingly validated that Arcturus is of low risk and proven effectiveness. Convincing validation is envisaged to involve measurement of software project personnel performance indicators, measurement of software development and maintenance costs, and demonstration that use of Arcturus significantly decreases measures of resources spent to attain comparable ends when compared to the use of predecessor programming environments.

SECTION 2

S h e l l s, G r a n u l e s, A t t r i b u t e s, V a l u e s, R P L s, a n d P D L s

In addition to supporting a concrete implementation language L, which provides executable representations of programs (that can be interpreted or compiled), Arcturus supports:

1. A rapid prototyping language (or RPL), and
2. A system of program design languages (or PDLs) at levels of abstraction above L.

The philosophy of program design languages follows that of Caine and Gordon [1975].

We take L and consider its shell. The shell of L consists of L with concrete data and operators replaced by function letters and constant letters which are uninterpreted, and thus have no assigned meaning. Shell programs are reinstantiated by substituting data and operations in new domains of interpretation, producing program designs at levels of abstraction above that of L. Such substituted shells provide possibly executable representations in other domains of operators and objects while retaining the control structures and syntax of L.

Consider now the program forms written in L and in the PDLs resulting from shell substitutions. The granules of these program forms consist of any of the syntactically well-formed units of the program forms such as constants, variables, operators, expressions, statements, blocks, modules, and so forth. (In short, the granules are the phrases of the program form with respect to the context-free grammar that defines the language in which they are written.)

Attribute/value pairs can be attached to arbitrary granules of program forms. [Many possible background representations are possible for this process, and different representations may be appropriate for different purposes. For example, an explicit list of dotted pairs may be attached to a granule, a hash link leading to a table of pairs may be used, a parallel file of pairs may be used, and so on.]

This general mechanism has many roles to play in the Arcturus environment:

1. Comments may be attached to granules using different attribute names, and these comments may be made selectively visible by different viewing "lenses", so to speak. What the application domain "expert" lens displays in its computed view may differ markedly from what the "programmer's" lens displays.

2. Given a design program in PDL, computer sizing estimates for time and memory consumption may be attached to granules of the program form by an interactive tool that develops a computer sizing estimate using a PDL design as its input.

3. Clocks and counters may be attached as attributes to granules during program interpretation to measure resource consumption of running programs. Display tools, such as histogram drawing programs, may access these attributes to compute pictures of the execution time profile of a program.

4. Release and version control attributes may be attached to program modules to designate which version is current and which version has been used to assemble the current system.

5. Status attributes may be attached to modules to portray their implementation condition. E.g. a module may be: designed but not coded, coded but not debugged, debugged but not independently tested, independently tested but not released, released but not integrated, or integrated into the running system. Management progress monitoring tools may check such status attributes to determine whether a sub-project is on schedule, and perhaps to perform exception reporting if anomalies are detected.

Programmer's Notebooks may assist individual programmers by computing task lists from such status attributes portraying what remains to be accomplished on a project.

6. Attributes may be attached to program granules and to files in the system database to record what system resources have been spent for use in cost accounting procedures.

Arcturus contains a <u>rapid prototyping language</u> (or RPL) which is a very high level, strongly extensible language useful for cheap, rapid construction of working prototypes. The RPL is a PDL in which it is possible to use statements and expressions in extensions of L.

In addition to the features of classical extensible languages, the RPL has several features that follow models established in LISP.

For example, in classical extensible languages it is possible to: (a) add a new declared data type, (b) introduce operations on the new data type and on its interactions with known types, including special appropriate syntax, (c) introduce notation to describe data constants of the new type, (d) print values of the new type in a user-defined format, (e) pass values of the new type to and from procedures, (f) assign values of the new type in assignment statements, (g) introduce nomenclature boundaries sealing off the internal names used in the definition of the type so they are not visible from outside, and so they do not interfere with identical nomenclature used elsewhere, and, in general, (h) endow the new type with all privileges accorded to built-in types originally defined in L.

In addition, the following ideas are borrowed from and modeled on capabilities in LISP:

1. Statements in the RPL can be "self-replacing" as well as value-returning, when evaluated (this being modeled after LISP FEXPRs and LISP MACROS). In this capability, arguments are passed to the defining body unevaluated, and quoted program text can be constructed, using nesting and "splicing". For example,

```
Macro Exchange (X) and (Y)
  '< declare
       temp: !GetType(X,Y);
     begin
       temp := !X;
       !X := !Y;
       !Y := temp;
     end;             >'
```

If we were to evaluate the call "Exchange(A[i]) and (B[x-2])," the above macro would produce a text fragment by filling in the template indicated between the quoted brackets '<...>'. Inside the quoted brackets, expressions of the form "!Exp" get evaluated to produce a fragment of program text which is substituted in the template in place of !Exp. For example, !GetType(A[i],B[x-2]) is evaluated using unevaluated arguments A[i] and B[x-2]. This looks up the compile-time type of elements of A and B (which, let us say, is Integer), and it returns the text "Integer" to use in the declaration. !X and !Y evaluate to A[i] and B[x-2] respectively. Thus, the following text results from calling "Exchange(A[i]) and (B[x-2])":

```
declare
  temp: Integer;
begin
  temp := A[i];
  A[i] := B[x-2];
  B[x-2] := temp;
end;
```

[Note: !X means splice in the value of X and !(X) means nest the value of X. For instance, if X := '<y + 3>' then !X*z ==> y + 3*z and !(X) ==> (y + 3)*z.]

2. Text fragments can be evaluated, as in Eval(E),

Lambda forms can be applied to argument lists, as in Apply(L,arglist), evaluation may take place in local contexts supplied by giving local association lists specifying pairs of formals and actuals, and the usual control can be exerted over the READ-EVAL-PRINT loop.

3. The underlying forms of L-programs are rendered as "keyword" list-structures (as in the representation of MLISP in UCI Lisp [Meehan 1979] or of CLISP in Interlisp) on which operations may be performed to enable program manipulating programs to be written. The results can be pretty-printed in the surface syntax of L. For example, powerful mapping functions, such as "Map (f) onto (A)" can be built up in the extension language by defining macros which produce program text in L which applies function _f_ to each node of a composite structure A.

Function calling forms are given an extended syntax in the RPL. In addition to the usual calling forms such as Place(a,b,c), one can have:

```
Place (a) on square (b) on board (c);
(PQ) is nonempty;
(X) is a member of (M);
```

in which the arguments, usually separated by commas, can be separated by constructions of the form ")w(" where _w_ is an identifier sequence separated by spaces, and in which initial or trailing identifiers can be present or absent. The system remembers each such calling form that has been entered and allows "automatic completion" in the following sense. Whenever a disambiguating prefix of a calling form is typed and the "escape" key is struck, the rest of the calling form is printed out up to the next parameter position or up to the next point of ambiguity. This feature is useful in typing long names of self-documenting code in PDL and RPL programs.

Using self-replacing calls of this nature, it is possible to write program transformations that extend the language in interesting ways, such as using objects of conceptually unbounded extent (such as iota infinity, for APL cognoscenti), as in:

```
Sum up (Map (Lambda(n)in( x**n/factorial(n)))
                 onto ( iota(infinity)));
if (W) is member of (N[i] union Vt)Star
then N[i+1] := N[i] union W end if
```

Finally, RPL programs may contain bracketed descriptions of operations, computations, relationships, and properties specifying what programs will do when they are later elaborated into concrete executable code in L. For example,

```
   {Let (PQ) be a Priority Queue;}
   {Let (T)  be a collection of input elements;}
   {Let (Q)  be an Output Queue, initially empty;}
begin
   {Rearrange (T) into a Priority Queue (PQ)};
  while {(PQ) is nonempty} loop
   {Remove the Largest Element in (PQ)};
   {Insert (it) on the Rear of the Output Queue (Q)};
   {Restore the Priority Queue Property in (PQ)}
  end loop;
   Return Q;
 end
```

SECTION 3

T h e P e r s o n a l W o r k s t a t i o n

The Personal Workstation of our current dreams consists of a system based on a powerful, cheap desktop computer and cheap flatscreen displays. We hope large, cheap flatscreen displays will soon make their appearance and that they will come in large rolls. Given this assumption, we would like to unroll some and cover a large desktop, and then unroll more and fasten it to a large area of a wall, say, the size of a blackboard. An independent portable keyboard and a portable stylus would connect (say, via radio) to the system.

We would like the flatscreen display to have the quality of liquid crystal --- that is, we want printed text, drawings, and figures to appear in it as if a black printed transparency had appeared inside a sheet of plastic or a pane of glass. We want the resolution to be sufficiently fine-grained to support high-quality pictures, line-drawings, and type-fonts of many sizes and styles. We want the flatscreen display to have independently switchable (x,y) coordinates so that subpictures and contents of windows can be incrementally changed, and we want the switching speed to be sufficiently fast that a page of text or picture can appear in a few hundred milliseconds.

The technology for driving such flatscreen displays has been known for some time, and involves windowing, clipping, inking, menuing, latching, and computation of simultaneous views in "panes" of a window. This technology has been developed progressively on the Lincoln Labs TX-2, on the Harvard PDP-1, on many Evans & Sutherland systems, and recently on the Xerox PARC Altos used in, e.g. Smalltalk and Interlisp. Cut-and-paste text editors, animation sequences (producing dynamic books), and a uniform command language using menuing, windowing, and simultaneous contexts (one of which is usually active and the others suspended), are well-known techniques of proven effectiveness for using such a medium.

We feel that the current technology, which uses a small display, is a disadvantage, causing users to operate as if they were manipulating many small slips of paper in stacks inside a shoe-box. Large flatscreen displays on desktops and walls should cure this ailment, making it possible to use normal size pieces of "electronic paper" and to have many of them displayed simultaneously on a desktop or wall, more like the properties of real paper.

In our dream workstation, we would like to see a graceful marriage of the worlds of paper and electronics. A very high resolution TV camera could shoot pictures of printed text, drawings, (or anything, for that matter), yielding images that could be shipped electronically. In addition, we envisage character recognizers that could process such TV images of text, producing descriptions of the characters and their fonts in "pre-runoff" form, suitable for text editing. We also could cut and paste pieces of electronic images resulting from TV capture of real paper images for inclusion in electronic documents. We would be able to use inking to include handwritten remarks in electronic documents to be shipped over computer networks. To go from the world of electronics back to the world of paper, we envisage good-quality, two-sided Xerographic printers (such as the Penguin variety of printer).

Given electronic books and manuals, we envisage programming the workstation not only to search via normal text editor search, but also to simulate browsing, via analog controls, starting at interpolated search points (as in searching for the name "Vickers" in a telephone book starting "somewhere near the end").

When we latch onto a piece of electronic paper lying inside our desktop and drag it along with a stylus, we envisage drawing a blinking frame along with the stylus as the stylus moves. When the stylus is depressed to unlatch and deposit the paper, the paper would get redrawn at the point of deposit as an overlay on top of whatever else was there below it.

It may be possible to program the workstation to do fancy things such as placing the text of a book on an imaginary football field somewhere to the left of the Moon, and flying out to view it at Warp 6, applying a spelling corrector to the football field, and having lights turn on everywhere there is an error (to get some portrayal of the error density, viewed from afar), and then zooming on selected areas for further text manipulation by normal means. However, we are not convinced that rapid, real-time, color displays (as in the E&S flight simulators) are necessary for our task. Real-time, color, simulated motion with peripheral detail, is probably unnecessary for our task. We include this remark in the discussion to illustrate the idea that we are not being as grandiose as we could imagine, and that we are not playing the game of, "whatever you have, I'll imagine something more general."

An important consideration for us, which we do not see strongly reflected in present work (though perhaps we are in ignorance), is the availability of prefabricated means for smoothly extending the user-interface language, using windowing, menuing, simultaneous computed views in panes of a window, command completion, and the like. We think it should be made easy to add a new class of menuing and display commands in exactly the same style that the original system supports. This encourages users to add tools whose command languages obey the same conventions as the original tools in the system --- providing a strong incentive to keep the user interface uniform in user created tools.

SECTION 4

T h e   I n t e r a c t i v e   P r o g r a m m e r ' s   N o t e b o o k

A computer can help managers and programmers by being an active agent that can remind, nudge, and report. Because of its persistence, it can assure thoroughness of adherence to prescribed disciplines. By this means, a computer can play a keystone role in assuring software quality in partnership with teams of people.

In Arcturus, there are envisaged several embodiments of these quality assurance ideas. Great care is exercised to engineer the human interfaces smoothly and not to be heavy-handed. Rather, the envisaged style is one of gentleness in the delivery of the computer's quality of persistence.

For example, by means of computerized checklists (the Arcturus Interactive Management Review Interview), version control, schedule management, and exception reporting, Arcturus uses the computer's capacity for management of detail to ensure thoroughness and thereby to increase the chances that effective management disciplines will be followed in the production of quality software.

Another example of the use of persistence is in feeding an undocumented program form to a document interview program which extracts comments interactively from a programmer as granules of the program form are enumerated at different control settings. The extracted comments are attached to program granules and can later be queried or displayed in computed views from various "vantagepoints".

In this section, we comment briefly on another of Arcturus' persistent, quality assurance tools --- the Interactive Programmer's Notebook.

When a programmer is handed a design, say in the form of a PDL program, he needs to manage a large collection of subtasks to implement it, optimize it, test it, and release it.

By feeding the Interactive Progammer's Notebook a PDL program, the Notebook will extract all the module names and will compute a status check list for each module, giving as many status categories as the user supplies. For example, the status categories may be: (a) designed, (b) coded, (c) debugged, (d) optimized, (e) tested, (f) independently validated, (g) released. Each module needs to go though these stages. When modules are coded so as to call on unwritten modules, the unwritten module names must be incrementally added to the task list, and their status must be monitored. Additionally, tasks and reminders may be added to the Notebook at will by the programmer (e.g., tasks such as creating test data sets, saving intermediate results on archive files for protection against crashes, writing progress reports, etc.). This is reminiscent of the personal "calendar" and "reminder" programs in existence on popular operating systems today. Time-critical reminders can be scheduled, and background processes that monitor for the occurrence of changes in the programmer's database can be set in motion periodically to report when their conditions are fulfilled (e.g., "Send me a reminder '90% spent' when my total user charges exceed 90% of my allocation in Sub-Project A6").

An emphasis is placed on automating the acquisition of status changes, e.g., from file extensions, or particular attributes attached by various tools. The Interactive Notebook monitors for the presence of attached attributes, such as an attribute saying that a module has passed an independent validation check, and updates its records automatically. Such automatic data capture avoids the disaster that would occur if all updates to the status records had to be made manually by the programmer --- a degree of tediousness that would likely defeat the advantages of using the system.

SECTION 6

S o f t w a r e   M a n a g e m e n t

In addition to the chronological phases of the software lifecycle, there are strands of activity that pervade the entire lifecycle. For instance, there are (1) management disciplines, (2) communication disciplines (including documentation), (3) training of new personnel, and (4) validation and quality assurance.

In a sense, (2), (3), and (4) are subordinate to "(1) management disciplines," since it is management's responsibility to assure quality at each stage, to train new personnel, to impose and monitor communication disciplines, and so forth.

In fact, much more comes under the heading of management, including (1) resource estimation, (2) lifecycle costing and accounting, (3) tasking and critical path scheduling, (4) early detection of poor performance, (5) initiation of corrective actions, and many other features of overall project organization.

Arcturus provides tools to support management of all these interacting strands of software lifecycle activities. Arcturus is designed to be a concrete realization of an entire software lifecycle support environment. Arcturus is designed so that its constituent disciplines and tools can interact smoothly, and so that we can validate that the concepts Arcturus embodies in fact improve programming performance.

The management tools in Arcturus utilize the computer's capacity for managing large volumes of detail with precision, and the interface between people and computerized detail management media is designed so that the human capacity to absorb and manage detail is not overwhelmed.

For example, since computers can keep track of large volumes of detail about such things as (1) What has been accomplished so far, (2) What remains yet to be accomplished, (3) What resources (time, people, dollars) have been used so far, and (4) What resources are estimated to be needed to accomplish what remains --- Arcturus tools construct and manage dynamically computed views about the status of this body of details which isolate from the huge volume under computer control views appropriate to management activities such as: (1) Critical path schedules, (2) Lifecycle cost reports and estimates, (3) Project task workbooks and checklists.

These views are periodically computed for human consumption and the body of data from which they are computed is incrementally updated and adjusted, perhaps by a mixture of manual and automatic means. Accounting data on cumulative resources spent to date may be automatically acquired. On the other hand, resource estimates and task completion notices may be manually acquired --- perhaps by interactive prompting and query of project personnel.

Arcturus can help assure software quality by reliance on the computer's capacity for enforcing thoroughness. For instance, consider a management discipline which relies on a collection of methods for assuring software quality including, for example:

(1) Early detection of poor programmer performance.

(2) Use of early prototypes.

(3) Effective documentation discipline.

(4) Proper training of new project personnel.

(5) Careful validation of software quality at each lifecycle stage using, e.g., design walkthroughs, independent validation of implemented modules (assuring performance goals are met by the use of independent test and validation teams).

(6) Flexible allocation of resources to tasks and organization of task structure to meet changing demands.

(7) Use of formatted debugging aids.

Poor or inadequate adherence to policy on any of these stated disciplines may incur expensive penalties in software quality, leading to well-known problems [Boehm 1973] such as late delivery, cost overruns, release of unreliable modules, etc.

The overall complexity of the task is such that under the crush of daily business, managers, analysts, and programmers may let certain details "fall through the crack," so to speak, witnout their being aware that they have overlooked (or perhaps thought about but failed to remember) one of the manifold aspects of proper performance of their assigned responsibilities. Also, there is a human tendency to cut corners under pressure --- such as omitting the implementation of formatted debugging aids before module testing (a probable false economy).

The deliberate or accidental omission of steps and duties falls under our definition of lack of thoroughness in adherence to selected software quality assurance disciplines.

By assuring thoroughness of adherence to prescribed disciplines, Arcturus management tools help assure software quality in partnership with managers and programmers.

SECTION 6

D o c u m e n t a t i o n

When we consider "programming in the large" (by large numbers of people, on large programs, with long maintenance lifetimes) versus "programming in the small" (by a few people, on a small program, with a short useful lifetime), we see that program documentation plays a critical role. In any large programming project that extends over many years, there is likely to be personnel turnover. This implies that new project personnel will have to read and understand the programs written by others. Furthermore, when programs get large, they also tend to get complex, or at the very least bulky --- in the sense that even though each microscopic patch of the program may be structurally simple when considered alone, there are many such simple patches packaged into submodules, modules, and larger program units, and there are many, many such units.

Another kind of complexity that tends to occur in large programs is that which derives from the inherent refinements of concepts in the application domain into executable underpinnings on the naked machine. Usually these refinements occur by means of one or more intermediate layers of representational media. For example, in an airline reservation system, application domain concepts might include flights, seats, dates, and schedules. At the level of the naked machine we have bits, bytes, and linear sequences of words. At intermediate representational levels, we may have files, strings, lists, pushdown stacks, queues, records, tree-indexes, and the like.

Each application system is a microcosmic example of a reductionistic system that reduces application domain objects and operations into executable combinations of machine primitives, and to understand a program at the level of the underpinnnings requires that we understand the higher level operations that are being mimiced by the low level mechanics. A critical function of documentation is to reveal the relationships inherent in the imitation of the highest level application mechanics by the organization of underlying representational media.

Another type of knowledge that may be critical to the understanding of a program is that which comes from <u>collateral</u> <u>reasoning</u> <u>systems</u>. These reasoning systems are those which do not participate directly in one of the layers of the reductionistic refinement system from the application level down to the naked machine level, but rather are those used to derive facts about one of the refinement layers or to understand why something at one of the layers works the way it does or achieves some desired effect. As an example, suppose we are computing a Fibonacci number. Three implementations are: (1) the implementation that runs in exponential time that computes Fib(n) by summing calls on routines Fib(n-1) and Fib(n-2), (2) the implementation that runs in linear time that stores an adjacent pair of Fibonacci numbers, sums them, and shifts the sum into the position occupied by the larger after shifting the larger into the position occupied by the smaller, and (3) the implementation that runs in logarithmic time which uses differences of powers of quantities derived from the golden ratio. To understand why this latter implementation works requires an excursion into the collateral reasoning domain of mathematics. It is often the function of documentation to reveal (or at least to point to) the relevant portions of an explanation for why something works using the agency of a collateral reasoning system --- and such a system may encompass many disciplines outside of computer science, such as geometry, kinematics, chemistry, optics, and an indefinite number of others independent of and not necessarily implied by the phenomena and laws of computation.

It is also noteworthy that documentation must play different roles for different audiences. Depending on the experience and knowledge of the reader, documentation should reveal appropriate facts --- what is appropriate to one reader may be either boring, obvious, and condescending to another, or completely beyond the intellectual grasp of yet another. Only a physicist may be expected to understand an explanation of why some computation works with regard to modelling of optical or kinematic phenomena. Only a programmer might be expected to understand mechanical details of nomenclature scoping for the dynamic lifetimes of certain program variables. Only an economist might be expected to understand a market elasticity computation, and so on.

Thus, documentation must deal with an indefinite number of domains of technical knowledge, with an indefinite number of classes of readers of varying sophistication and technical preparation, and with relationships between layers of different representations spanning the gap from the naked machine to any of an unbounded range of application domains. The creation of good documentation to serve these multiple purposes probably requires a considerable capital investment, and can probably be justified only for programs with long maintenance and upgrade lifetimes for which the savings realized by good documentation at least reimburse their cost of original development.

While this introductory discussion points to the importance and complexity of good documentation, computer science has yet to produce a good theory of documentation and to give us effective means of organizing it. In fact, we don't even have a good theory of program comments --- one that rises above the level of aphorisms and one for which it has been experimentally validated that it, in fact, improves measures of programmer performance.

It is partly this state of appalling ignorance, partly the key importance that a valid theory or method of program documentation would have, and partly the challenge of wanting to make strides in this important field that leads us on the quest we are trying to initiate in our initial philosophizing about Arcturus.

Some initial ideas we have on documentation are as follows:

1. Paper is the wrong container for documentation: One cannot write down on a printed page all that needs to be said in adequate documentation without causing a great deal of clutter. Further, it is difficult for a given expert to extract the relevant from the irrelevant in such a medium. What is needed is a database in which program forms at various levels of refinement have attributes attached to their granules leading to comments, whereupon various computed views can make these comments selectively visible.

2. Dynamic prompting as a method for encouraging ease of construction and completeness of coverage of comments: We envisage a tool that would accept an undocumented program form (or a pair of forms consisting of a program and its refinement), and which would ask questions about the program form,

using a level of granularity and a subset of questions determined by setting controls in the tool. The answers to the questions would be attached as comments to be made selectively visible by computed views or queries.

3. Computing Translations Using Translation Tables and Annotation Substitutions: When a program form is displayed in a pane of a window on the Arcturus desktop, it is possible to display other computed views of the program in other panes of the window simultaneously. Some of these views can consist of annotated and/or translated views of the program. For example, suppose we have a translation table that maps phrases as follows:

```
not (X = Ø)   ==> X is not empty
Largest(X) ==> Extract Largest Element in (X)
Enqueue(x,Q) ==> Add Element (x) to the
                       Rear of Queue (Q)
```

Then, using these transformations to translate the following piece of code:

```
while not(PQ = Ø) loop
   x := Largest(PQ); ReHeapify(PQ);
   Enqueue(x,Q);
end loop
```

We can derive the following computed view containing self-documenting descriptions, with respect to the above translation table:

```
while PQ is not empty loop
   x := Extract Largest Element in (PQ);
   Reheapify(PQ);
   Add Element (X) to the Rear of Queue (Q);
end loop
```

In addition, when we have performed statements attaching attributes to variables, as in declarations or in statements such as:

Let (PQ) be a Priority Queue

We can call for a transformation that annotates program variables with their declared or attached descriptions. An annotation might replace the first instance of a variable PQ with an annotated text such as {PQ: a Priority Queue}. This could be used to remind or inform the user of the properties and purposes of a variable in a program text.

SECTION 7

I n t e r i m   E n v i r o n m e n t s

We intend to approach the evolution of Arcturus by means of a number of interim steps. We envisage the construction of a number of interim environments that will become progressively more complete, refined, and efficient.

Our first interim environment will likely be a prototype implemented on a PDP-1Ø in UCI LISP [Meehan 1979], with only a modest subset of the capabilities we have sketched in this paper. Later interim environments may take advantage of desktop computers and flatscreen displays if they become available.

The scope of Arcturus is sufficiently broad that it may take a decade or more to complete the construction of a production quality Arcturus.

R e f e r e n c e s

Boehm, B.W. [1979], Software Engineering as it is, 4th Int. Conf. on Softw. Engr., Munich, Germany.

Boehm, B.W. [1973], Software and Its Impact: A Quantitative Assesment, Datamation, (May 1973).

Caine, S.H., and Gordon, E.K., [1975], PDL - A Tool for Software Design, Proc. NCC, AFIPS Press, Montvale, N.J.

Meehan, J.R. [1979], The New UCI Lisp Manual, L. Erlbaum Assoc., Hillsdale, N.J.