SOFTWARE ENVIRONMENT RESEARCH: DIRECTIONS FOR THE NEXT FIVE YEARS*

by

Leon Osterweil
Department of Computer Science
University of Colorado, Boulder
Boulder, Colorado 80309

CU-CS-195-80 October, 1980

* This work supported by National Bureau of Standards
Contract #NB80NAAD8714

Abstract

This paper addresses the question of how best to expedite progress over the next few years towards the goal of making general purpose software environments widely available. The paper begins with a discussion of the meaning of the term environment and some conjecture on the expected benefits of environments. Next, five characteristics which distinguish a true software environment are enumerated and discussed. It is suggested that progress towards generally available general purpose environments can most effectively be made by studying these five characteristics, their interrelations, and the issues involved in realizing systems displaying them. It is then proposed that the indicated studies are primarily experimental and observational. Hence a program of experimentation is strongly endorsed as the most effective way to facilitate widespread creation of environments. Some specific proposals for this experimental research are then advanced.

I. Introduction and Background

During the past decade there has been a growing realization that the multitude of tools fashioned to facilitate software development and maintenance has by and large failed to produce the desired effects of reduced cost and improved quality. Considerable investigation into the cause of this failure has been carried out, and there now seems to be good agreement that it is in large part attributable to the inability of most tools to provide substantial close support to software practitioners in their actual work. Most current tools and tool systems focus their support on narrow aspects of software processes such as editing and testing, leaving other areas unaddressed and unsupported. Further, most tools oblige the user to accomodate himself or herself to the idiosyncrasies and underlying philosophies of the tools and their creators, rather than vice versa. As a consequence many potentially useful tools have languished unused.

Clearly, this situation could be improved if tools were lashed into configurations, continuously supportive of the user in his or her actual day-to-day work. Such a configuration of tools has come to be referred to as a Software Environment. Although the desirability of such environments and their benefits have been widely discussed, there has to date been relatively little organized research or actual implementation in this area.

Largely in the interest of encouraging the growth of research and development activities in the area of Software Environments, the National Bureau of Standards Institute for Computer Science and Technology held a workshop on the subject late last spring. Workshop participants were divided into four discussion groups to consider

different aspects of the problem of facilitating the growth of this work. The author was priviledged to serve as a discussion leader for one of these groups. The topic considered by our group was the direction which Software Environment research ought to pursue over the next five years. What follows here is a personal summary of the conclusions reached and recommendations made by the group. The purpose of this paper is to acquaint wider audiences with what should be considered an area of great and growing significance for the software community.

II. Software Environments — Their Nature and Expected Benefits

Before embarking on deliberations about research on software environments, it seems clear that it is necessary to first address the central question of what a software environment really is. This seems to be a particularly difficult and risky endeavor because this research area is still too young and speculative to lend itself to any extensive formalism. Indeed too much formalism and rigor at this stage might serve to discourage and thwart useful seminal work. On the other hand, because there does seem to be considerable intuitive agreement about some specific instances of software environments, their general nature might perhaps best be conveyed through several examples. The following is a list, intended to be representative rather than exhaustive, of software scenarios for which total software support systems should be considered environments.

- A large group (> 500) of programmers is charged with the maintenance of long lifecycle (> 10 years) software which they, in general, have not written themselves. This group needs (at least) tools to help study, and analyze the code; maintain

version control over the code; and test and verify changed code.

- A team of real-time programmers is in the early stages of analysis and design for a distributed real-time processing system. The team needs tools to facilitate the process of understanding and agreeing upon the system requirements, creating, adjusting and verifying the system design, coordinating and controlling the system coding and testing, and assisting in the process of altering system code, design and/or requirements whenever this is indicated.
- A management team is responsible for the software production efforts of up to several dozen people, possibly working on several different projects. The team needs to make crucial resource allocation decisions in order to assure timely and acceptable completion of the projects. The team needs tools and mechanisms for obtaining visibility into the activities and progress of the software production personnel and projects.
- An independent team of quality control analysts is charged with the responsibility of determining the presence or absence of errors from finished software products in a timely and cost-effective fashion. They need a collection of analysis and testing capabilities for proceeding in a straightforward way to an identification of specific errors, or for raising, in an orderly, systematic way, their confidence in the absence of errors.

In each of these scenarios it is clear that individual tools and aids do exist to facilitate the work, but that these tend to be isolated,

stand-alone tools capable of supporting only isolated parts of the work to be done. What is needed, however, are tool configurations to provide integrated, continuous support.

Hence it seems the essence of an environment is that it is a software system which attempts to redress the failings of individual software tools through synergistic integration.

It is necessary to elaborate on this statement by being more specific about the nature of the "failings" of individual tools and what is meant by "synergistic integration." Each of the four scenarios just presented can be relatively easily described by a word or phrase characterizing the nature of the underlying software job to be done. Thus the first is a maintenance scenario, the second a large-scale production scenario, the third a management scenario, and the fourth a Quality Assurance (QA) or Verification and Validation (V&V) scenario. An environment is characterized and distinguished by the fact that its specific mission is to help human individuals and teams to perform their software jobs more effectively. Insofar as a support system's tools, interfaces and internal structures are integrated to provide strong continuous support for a specific job at hand, the system can be considered to by "synergistically" integrated to meet the needs of that job, and hence to be an environment. Individual tools "fail" because their support, although often strong, is usually only narrowly focussed on isolated aspects of a software job.

It now becomes clear that the task of creating effective environments, which has proven to be an elusive and difficult one, is so elusive and difficult because it is tantamount to understanding the nature of the fundamental software processes themselves. A specific environment

does not merit the name unless it provides strong uniform support for the entire process it is intended to facilitate. That is not possible unless that process is fully appreciated and understood. Hence in a fundamental sense, environment development seems destined to progress more or less in parallel with a growing understanding of underlying software processes themselves.

Presently the software research community seems to be in a good position to make substantial progress along these parallel lines. We have grasped the critical importance of software development, maintence, management and quality assurance. We have begun to differentiate between them and yet also to see their close interrelation. We have constructed significant individual tool capabilities in these areas and gained valuable experience (both good and bad) with them. Partially through these experiences we have come to better understand the underlying processes which are in need of support. We have defined procedures and methodologies to guide humans in these various software endeavors, and have begun to gain experience with them. In summary, we seem poised to attempt to meld together the most promising of our tools into systems of support for our most promising methodologies in the key areas of software production, management, maintenance and V&V.

Although optimistic about current prospects for making significant progress in this work, it also seems most appropriate to fully appreciate the immensity of it. This work seems at least as ambitious and difficult as the work entailed in creating superior high level programming languages.* In terms of progress, we seem to be at a very preliminary stage corresponding to the period immediately prior to the appearance

*This analogy was first presented by Donald Good .

of Fortran in the mid-1950's. What lies ahead for us is a long period of development and experimentation during which we must match automated tools and problem expression media with developing understanding of underlying problem areas and emerging solution methodologies. It seems clear that this matching process will proceed through a sequence of successive improvements in the form of a succession of environments, much as high level language technology has improved through the creation and evaluation of a succession of high level languages. It seems, moreover, that the analog of Fortran (i.e., a widely applicable and acceptable general purpose environment) has yet to appear.

Before embarking on a costly broad scale program of software environment research and development, it also seems essential to reflect on its probable benefits. There seems to be a unanimous feeling that good software environments will raise the quality of software while lowering its overall lifecycle cost. There is less agreement, however, about the specific ways in which these goals will be attained. Increased manageability and maintainability through improved visibility are widely expected. More effective testing and analysis throughout the software lifecycle are also projected, as is greater care and thoroughness during the critical early stages of software development. All of these will certainly effect greater quality and reduced overall cost, when the total lifecycle (including maintenance) is considered. Yet there seems to be disagreement about which are the most overriding and crucial of these.

Here too, the high level language analogy seems useful. It is difficult to determine even now, no less in 1950, the greatest benefit of a high level language. Is it readability, manageability, increased

productivity or improved quality? It seems clear that high quality, general purpose environments will offer ample quantities of all of these benefits. It is suggested* that these benefits will probably accrue indirectly, because environments will relieve software people of much drudgery, freeing them for proportionately more creative and intellectual activity. Thus software solutions will be more carefully thought out and arrived at after more numerous and thorough iterations.

III. Overall Strategy for a Research and Development Plan

Having established at least a general notion of what software environments are, and why they are important, it is reasonable to next consider what needs to be done in order to facilitate their creation.

The preceding discussion has established that the essence of a software environment is synergistic integration of tools in order to provide strong, close support for a software job. Hence it seems reasonable to proceed by examining in detail what that essence specifically implies and entails. Our discussions led to the conclusion that there are at least five (not necessarily orthogonal) characteristics which a support system must possess in large measure if it is to merit the appellation "environment." It seems therefore as a consequence that any organized program of software environment research and development should be focussed on studying these five characteristics: their nature, their achievability, the ways in which they might possibly conflict with each other, and, eventually, whether they are actually the critical characteristics. The five are:

---

*by Donald Good

1. Breadth of Scope and Applicability. An environment must extend strong support to a software person or team across the full range of the software job being done.

2. User Friendliness. An environment must provide strong, direct comfortable support. Thus it must not oblige the user to accommodate himself/herself to it, but rather it must accommodate itself to the user. This accommodation must extend beyond the usual items: clear diagnostics, fail-soft error recovery, easy-to-use input languages, and HELP subsystems. The environment must in addition provide direct, painless support for the user in the actual procedures of his/her day-to-day work. It must not oblige the user to adapt to or relearn a new way of doing business.

3. Reusability of Internal Components. An environment must be flexible in adjusting to different and changing user needs. This flexibility can probably only be achieved by constructing the environment out of tool fragments, rather than whole tools. A collection of monolithic tools, standing side-by-side under the umbrella of a common user interface, is unlikely to be both flexible and efficient in meeting the possibly unforeseen diversity of needs which users may have. A comprehensive collection of easily reconfigurable tool fragments would offer this flexibility with the potential for efficiency as well.

4. Tight Integration of Capabilities. An environment's capabilities must work closely with each other to provide the user with a sense of continuously strong support. An environment must support a user community in doing its work according to its own procedures.

Yet the environment itself is to be implemented by a possibly small set of tool fragments to be configured and reconfigured in response to the (possibly changing) requirements of the end-user community. This poses the danger that the user might be made uncomfortably aware of the fact that his/her needs are being met by an amalgam of different tools and tool fragments. This must be strenuously avoided, as it violates the principle of User Friendliness. It can be naturally avoided by assuring that the individual tools and fragments maintain an awareness of the existence and capabilities of each other. Through this awareness the tools, tool fragments, and integrating software should avoid duplication of services and reports. The tools and fragments must also be preconditioned to uncomplainingly tolerate each other's quirks.

5. Use of a Central Information Repository. It is quite reasonable to think of an environment as an information utility. In an important sense, the purpose of an environment is to assure that software workers can get the information they need to do their jobs at the time the information is needed. From this perspective, the purpose of an environment's tools is to capture such information, analyze and process the information, and disseminate the information. Given this view, it is reasonable that an environment should actually be implemented along the lines of this model. At the center of the environment must reside a data base of total information about the software project. Surrounding this data base should be an information management system whose job it is to access the data base in response to requests made by the environment's tools, tool fragments, and user interface components.

It is important to reemphasize that these five characteristics are not represented to be orthogonal, nor is it suggested that they ought necessarily to be orthogonal. We do represent that they capture the essence of what an environment ought to be. Hence in that they overlap or conflict, that overlapping or conflict presents a possible obstacle to the eventual effective construction of general purpose environments, and thereby suggests an important area of inquiry and research. Some of these areas of apparent conflict will be identified and discussed in some detail in subsequent sections.

IV. What Must be learned from a Research and Development Program

This section considers each of the five distinguishing characteristics of an environment. For each the nature and importance of the characteristic is elaborated, with respect to the subject of software environments. The central questions which must be explored in order for true general purpose software environments to become realities are then explored.

1. Breadth of scope and applicability.

The central issue here is the need to determine how broad and encompassing an environment can reasonably be expected to be. It has been observed, only half in jest, that "there must be something like $2^{1,000,000}$ different environments," that might be built. These differ from each other along a multitude of coordinates which one might use for categorization, and, indeed, in the coordinates which are appropriate.

The representative list of examples of support systems, given in the first section of this report, begins to indicate this diversity. That list indicated that it is reasonable to consider building

environments to support software development, maintenance, verification, testing and management. Other software activities worth considering are documentation and distribution.

Within each of these activities there is considerable diversity in what might be supported. For example, a software production environment might have to support any particular software lifecycle model or concept. Thus it might encompass some or all of: requirements analysis, preliminary (architectural) design, detailed (algorithmic) design, and coding. It would presumably also support some level of testing, analysis and verification. This support might be applicable only to the output of the coding phase or to any or all other phases. The environment would have to generate reports, summaries and analyses upon which to base the various reviews called for by the lifecycle model as well. Similar broad variation should be expected among all support systems which might be considered environments for facilitating the various other software jobs.

More variation must be expected as a result of differences in source languages. A verification or maintenance environment for COBOL programs must of necessity be different from one for HAL/S programs. There is a certain amount of obvious truth to that statement. The issues can become much deeper, however, when one considers the remarkable range of programming languages and the attendant effects they have on support environments. A language such as LISP is different from COBOL or FORTRAN in some very fundamental ways. Some of these differences make it possible, indeed natural, to edit, test and analyze LISP programs in elegant and powerful ways which would be impossible for a language like COBOL. The INTERLISP system, for example, [Teit 78]

exploits this, giving a tangible example of the profound impact that a language can have on its support environment.

In a similar way, an environment to support EL1 [ECL 74] program production would have certain fundamental differences from environments for most other contemporary languages. EL1 supports the design phase, as well as the coding phase, of software development. Hence testing and certain verification procedures can and should be applied uniformly to designs and code in an EL1 development environment (as is currently acutally being done [Ploe 79]). The close confederation of these phases, on the other hand, makes it more difficult to separate and identify progress on these phases. This could complicate matters in the creation of an environment for the management of EL1 software development.

Different application areas must inevitably also lead to differences in the environments needed to support them. For example, Fortran might be used to create a numerical software library or a spacecraft control system. In the former case there would typically be little or no formal requirements analysis and preliminary design. This appears to be due to the maturity of the problem area and the suitability of mathematics as a requirements and design notation. In the latter case there would be extensive amounts of requirements and design creation, analysis, review and reporting. Clearly the environment's support mechanisms would have to vary similarly. The latter problem area is also generally considered to be of more critical importance than the former, as errors have the clear potential for causing loss of life and property. Thus a testing and verification environment for spacecraft control would of necessity include costly verifiers and simulators

which would probably be inappropriate in a numerical software testing and verification environment. Indeed, because spacecraft control software is concurrent, tools for testing, analyzing and verifying the concurrent behavior of this software would be essential here, but of no value for the numerical library. Different problem areas also mandate the need for differences in security mechanisms, version controls, and customer reporting in environments supporting these problem areas.

A project's size can also have an important impact on the support environment for that project. Here the primary effect seems to be the need for better and more effective communication and control as project size grows. The communication needs of a 2-3 person project are obviously far more modest than the needs for a 100-person project, involving 2-3 levels of management. In the larger project there are also needs for privacy and configuration management and control which are either absent or sharply reduced in a small project environment

Having thus established that there is a need for an enormous number of different environments we are now left with the question of how to supply them all.

Some questions which seem worth exploring as vehicles for elucidating this overriding question include the following:

- Under what circumstances, if any, is it reasonable to synthesize larger, more encompassing environments out of smaller ones?
- Along what degrees of freedom, if any, can we expect to transform one environment for use in meeting a related set of needs. (e.g., it seems reasonable to build tool modules which

could be altered to change a PL/I production environment to a FORTRAN production environment. What other sorts of alterations can be made?)

- What sorts and amounts of methodological change in a using project can be comfortably supported?

Although a certain amount of this inquiry seems self-contained, the answers to these questions must certainly come, at least in part, out of the research into the other four characteristics of an environment, to be described next.

2. User Friendliness.

As noted earlier in this report, an environment must present its repertoire of support capabilities to its users in as supportive, unobtrusive and non-interferring a way as possible. There are a few ramifications of this basic requirement that bear elaboration. Most fundamentally, the underlying tool capabilities must be robust enough to survive user abuse (intentional or inadvertent), communicative enough to both explain errors in use and instruct in proper use, and liberal enough to both accept user input and produce user output in a form and language close to that of the software activity being supported. Individual software tools invariably suffer disuse and distrust for lack of one or more of these essential characteristics. Hence it is all the more important that constituent tool capabilities all be robust, communicative and colloquial.

User friendliness in an environment, however, entails more than just assuring the friendliness of individual tools. In addition the accessibility and usability of the entire package of tools must also be assured. Thus, for example, there must be adequate mechanisms for

acquainting the user with the range of capabilities available and guiding in the selection of appropriate capabilities. This need to keep an accurate catalog of available capabilities seems clear. What is less clear is whether the catalog should be used as a basis for attempting to reduce duplication of capabilities. Experience suggests that a sort of Software Darwinism can often cause better capabilities to automatically supplant weaker capabilities without the need for external enforcement. On the other hand overly large, confusing ensembles of tools can be sufficiently intimidating to the user as to discourage the use of an environment.

Regardless of the size or sophistication of any tool cataloging or indexing scheme, it is essential that the scheme, and the underlying capabilities themselves, be able to communicate with the user in a way with which the user is familiar and comfortable. The central issue here is that the purpose of an environment is to support the user in performing his/her job. This communication between the user and the environment must be in the terminology of the user's job setting. Support capabilities extended must directly support the methodologies and institutions of the user's job setting, rather than forcing the user to alter working or thinking habits in order to use the environment capabilities.

In some sense what is being described here is not simply friendliness to the user, but rather friendliness to the user's way of doing work. This appears to be more easily demanded than furnished. As already observed, if we are to be saved from having to recreate every support system and environment from scratch, it will be necessary to configure environments as much as possible from standard modular

capabilities. We now recognize that this configuration must be done in such a way as to directly support the user's way of getting his/her job done, no matter what that may be. The user's procedures should, in addition, be expected to change with time. The support environment must likewise change accordingly while remaining supportive and friendly to the new procedures. This appears to require the use of extremely flexible, robust, and compatible modular capabilities. It will be necessary to determine whether it is reasonable to expect to be able to build such modules which, nevertheless, display acceptable efficiency characteristics.

Also important to this line of inquiry is the question of how artificial intelligence and human factors research is applicable to investigations of user friendliness in environments. This is apparently a fertile area of exploration, whose most obvious aspect seems to be the way in which computer graphics might prove useful in helping to achieve friendly user interfaces. Graphics should be expected to be particularly useful when the problem area and/or its procedures can be naturally captured pictorially. Thus, for example, a software management environment would presumably profit from being able to communicate with its users by way of management procedure diagrams, time and effort graphs, PERT charts, and specimen report forms. Similarly a requirements or design creation aid within a software development environment would presumably profit from being able to directly display the pictorial specifications which are the natural form of SAMM [Step 78], SADT [Ross 77], or RSL [Bell 77] specifications.

It is less clear, however, that the use of graphics will be of much help in environments supporting communities where pictorial forms of

communication are not already in use. In addition, there is considerable doubt that the expense of elaborate graphics systems such as those featuring color and motion will prove to be justified.

3. Reusability of Internal Implementation Modules

Earlier sections of this report have already discussed the apparent need for an environment to be constructed out of small flexibly rearrangeable modules, or tool fragments. This appears to be perhaps the only way in which we can expect to construct a number of different environments without having to build each from scratch. It appears to be as basic an idea as the manufacturing notion of building and maintaining a product line (e.g., automobiles, TV sets, appliances) out of a modest set of standard parts and subassemblies. We have also already observed that this notion appears to complicate efforts at making environments user friendly. That goal seems to require the total concealment of the identities and characteristics of the implementation modules, and their smooth welding into a support system patterned closely after the user's own procedures. This appeared to place very heavy demands on the flexibility and interchangeability of these modules, suggesting perhaps that they must each be very narrow in scope if this goal is to be attainable.

The foregoing discussions lead one to believe that the "Internal Reusability" characteristic might not be so much an independent characteristic of environments as, perhaps, a derivative of other characteristics.

Be this considered a derived or independent environment characteristic, there appears to be no doubt that investigation of the feasibility of building environments out of small tool fragments is

one of the most important research areas for the near term. Certainly if experience does not show that significant families of tool fragments can be assembled and found to be flexible, broadly applicable, yet efficient, then it will be necessary to drastically revise our thinking about the practicality of creating a diversity of user-friendly software environments at bearable cost.

There is little difficulty in identifying useful tool fragments for use in certain places of some environments. For example, a parser seems to be a good example of a useful tool fragment, as a number of tool capabilities rely upon a facility for creating a token string or parse tree. Thus a single parser would be used by a variety of subsequent tool fragments for doing such things as prettyprinting, error checking, static analysis, or compiling.

The parser would perhaps be coupled with various of these subsequent tool fragments in different environments and at different times. A parser is a particularly good example of a tool fragment, because parsers can be automatically created by parser generators, and hence very readily altered as well, for example to meet changing needs for different languages and dialects. Hence here is an example of a tool fragment creation mechanism which is very flexible in creating a powerful, reasonably efficient, widely applicable tool component.

This example is encouraging, and it serves to stimulate looking further for other such tool fragments. It also appears that a set of general purpose static analysis modules would constitute a good tool fragment. These modules would implement a small number of widely applicable data flow analysis algorithms, operating only on annotated representations of program data flow. These representations would be

created by other tool fragments as abstractions of the original program. Hence the data flow analysis fragment would have little knowledge of extraneous source language detail. Research is showing that a small, well chosen set of data flow algorithms can be useful in error detection, verification, and optimization across a broad family of source languages. Research also appears to indicate that a concise pseudo-language notation can be used to effectively direct the automatic configuration of the algorithms into the various specific error scanners and verifiers that might be needed in different environments or as the needs of a given environment evolve.

Promising as these examples seem, there is nevertheless a feeling that they are rather isolated. There is, for example, pessimism about the existence of similar tool fragments to be used in building such important environment components as user interfaces, management reports, generalized editors and graphics packages. Such fragments can probably be built, but they will have to be the products of research and experimentation still to come.

The need for tool fragments to maintain a central data base is perhaps the most pressing need of all, yet it is also perhaps the most controversial. Although there is some sentiment that acceptable fragments of this sort already exist, another point of view holds that existing information management system capabilities might prove to be too inefficient to be an acceptable part of a full environment. Unfortunately, the full range of requirements placed on an information management system by an environment are not yet known. They must be determined in order to enable definitive study of this crucial area.

4. Tight Integration of Tool Capabilities.

A true environment is characterized by the close interaction and cooperation of its constituent capabilities. This issue has already been touched upon in discussing the ramifications of the term "user friendly." In that discussion we stressed that user friendliness specifically implies friendliness to the way in which the user does his/her job. Thus tool capabilities must be merged into a system offering smooth continuous support for the user in the performance of actual work procedures. Clearly this requires at least the appearance that the constituent tool capabilities are working closely together. In a true environment this close cooperation among capabilities must be actual, not simply apparent.

In a smoothly functioning environment it will be important for the tool capabilities to be aware of each other. Thus tools should facilitate each other's work by pre- and post-processing data structures for each other. Tools should also be careful not to duplicate services and messages to the user. In these ways the efficiency and overall appeal of the environment to the user are enhanced. INTERLISP is a prominent example of a support system whose constituent capabilities are very tightly integrated both in fact and appearance, to yield a very appealing system.

We have already discussed the fact that the need for tight integration appears to pose a direct conflict with the need to construct environments out of flexible, general, reconfigurable modules (tool fragments). It seems clear that if the tool fragments are too general and reconfigurable, then they cannot exploit any significant knowledge of each other's workings — only each other's interfaces. Thus it seems

close integration can only be projected as an illusion by such components as the information management system or user interface.

Before resigning ourselves to the apparent irreconcilability of these two characteristics, however, it seems useful to study the INTERLISP example. One important motivation cited for needing to build environments out of tool fragments was the need to alter environments to fit a range of (probably evolving) user procedures. We find, however, that INTERLISP seems to be able to accommodate itself to a range of (changing) user modes. Thus apparently this can be accomplished with tightly coordinated tool capabilities.

It can be argued that INTERLISP is still rather narrow in its range of support and user community, thus probably not qualifying as a true environment, and that it is only this restriction in scope that enables it to be both flexible and tightly integrated. It seems, however, that this argument simply supports the proposal that much can be learned by studying INTERLISP and attempting to extrapolate from this example.

5. Use of a Central Data Base

The final, and perhaps most important, characteristic of an environment is that it be coordinated and focussed by access to a central repository of information. It is widely proposed that a software project is profitably thought of as being a coordinated effort to gain and disseminate a highly structured body of knowledge about a problem and its solution. That being the case, the progress of the project will be best assured and facilitated by capturing, structuring and disseminating that body of knowledge as faithfully and effectively as possible. These considerations seem to clearly imply the use of a data base and encompassing information management system as the

centerpiece of any environment.

Clearly, by taking this approach the effective diffusion of knowledge to all project personnel is facilitated. This certainly does not imply the giving of all pieces of knowledge to all people at all times. Quite to the contrary, effective diffusion of knowledge means purveying to each person precisely that information which is needed to accomplish his/her job at any given time. This would be accomplished by using the environment's tools to access the data base for specific data in response to needs as expressed to the tools by users. Tool capabilities might simply search the data base for needed information, might report back combinations of data or data aggregates, might update the data base in response to user input, or might augment the data base with the outcomes of analyses of data base contents as requested implicitly or explicitly by environment users. In all cases the outcome would be an up-to-date, centrally accessible body of complete project information, whose access would be facilitated by the tools of the environment.

This hightly attractive picture seems to be marred by serious questions of procedure and practicality. The most immediate questions seem to be questions of what should go into the data base and how it should be organized. The immediate and obvious answer, that "...everything should go into the data base...," is obviously simplistic and unuseful. There are widely different opinions of what is meant by "everything." For example some people believe that a software production environment should preserve in an archive all obsolete versions of code, all discarded designs, even all of the sketches and jottings produced during early problem formulation. Others object to this,

stressing the lack of utility and inevitable large-scale waste of resources inherent in this approach.

A resolution of this issue seems to come out of consideration of the optimal structure and organization of the data base. It is proposed* that the data base be organized as a model of the software activity being supported. In this approach the users, processes, data items, data flows and procedures of the software activity and setting are modeled and represented as entities, attributes and relations in a data base, managed by an information management system. Thus the data needed by an individual is readily available because it is grouped within the data base as the attributes and relations of a small set of entities. The need for analyses and reports can be semi-automatically identified and satisfied as the result of recognizing when entities, attributes, and relations within the data base have no current values. Tools could be invoked (manually or automatically) to supply these values. Experience shows that this approach has been widely used with good success.

Another important concern is to determine what procedures are necessary to insure the correctness and consistency of the data base, especially in the face of the continual changes to which it will be subjected. The magnitude of this problem is perhaps most graphically illustrated by considering the impact on a highly structured software development data base of such an apparently small change as altering a single line of program text. In particular, if this source line is a declaration statement, then its alteration might render invalid parts or all of such related data base elements as the token strings, parse tree, flow graph, and diagnostic reports. For each change, all possibly

---

* by Daniel Teichroew

impacted data base objects must be known, then analyzed, then perhaps purged or altered. The potential cost of such activities is intimidating, yet the necessity of these activities is undeniable. It may very well be that consideration of the need and cost to do this sort of updating will be a key factor in determining the size of data bases for support environments.

V. A Five-Year Research Plan

The purpose of this section is to suggest the outlines of a possible coordinated strategy for conducting a research program aimed at providing substantive answers to the questions posed in the previous section. This strategy was arrived at after careful consideration of both the knowledge needs, as just described, and the current state of the experience and expertise of the research community. It was noted that most of the learning needed is empirical and pragmatic in nature. A great deal of qualitative experience and quantitative statistics must be accumulated. Further it was noted that a number of researchers are currently poised to begin experimentation aimed at accumulating some of the needed knowledge. Thus, our group concluded that it would be most effective for these and other researchers to embark upon a program of experimental work which is guided, at least in a general way, towards the objectives which we have agreed upon as being desirable.

Accordingly, our plan essentially maps out a program of experimentation, having two separate thrusts. The first experimental thrust, intended to commence immediately, calls for the study and development of prototype support systems, each of which is intended to provide some specific insights into environment characteristics, as well as experience with moderate scale tool integration.

The second experimental thrust, not intended to begin for perhaps three years, will use the experience gained to attempt the synthesis of some full scale general purpose environments.

The studies of prototype systems during the first thrust should each be aimed at learning about one or more of the five stated characteristics of an environment and their interplays and relationships. It seems that the research community is currently in a position to learn a great deal by studying existing support systems and frameworks and also by building a variety of new, small to moderate scale support systems. The study of existing systems is a particularly logical step in that it should provide insights into effective integration strategies as well as answers to questions in some of the five previously described areas. Such studies should also make clear the areas in which more learning is most needed and in which this can be accomplished through new system construction. Thus, clearly, this system construction should be done in such a way as to elucidate as many of the outstanding critical questions as possible. Clearly it is possible now to gain a considerable number of insights into such questions as:

- How should environment data bases be structured and maintained?
- Upon what sorts of tool fragments might environments be built?
- What are some specific tradeoffs between flexibility and tight integration of tools?
- What are reasonable uses of graphics in user interfaces?

Following shortly are some suggestions for research projects aimed at providing insights into some of these and related questions. These suggestions are intended to be taken as examples rather than mandates.

It is expected that as this line of experimental research proceeds, the level of tool integration will increase, forming the basis for the second research thrust -- namely large scale integration of tools into general purpose environments. This thrust, though properly based upon the first thrust, is expected to have a different character, focussing mainly on the issue of breadth of scope. It is expected that this line of experimentation will draw upon all previous experience and learning, in attempting to determine the reasonable limits of tool integration. The construction of an environment so powerful and encompassing as to meet the changing needs of all people at all times will certainly be found to be an unreasonable goal. It is reasonable, however, to expect the limits which ought to be placed on the generality and scope of general purpose environments, as well as the techniques which are helpful in achieving large scale tool integration.

Our group agreed that it is important to delay this second thrust for 3-5 years, rather than initiate it immediately. There currently seem to be so many important gaps in our knowledge in critical areas, that this sort of enterprise seems currently to be inordinately risky. It was felt that any research activity, initiated now, and aimed at such large scale tool integration would rapidly bog down amidst the crossfire of the critical unresolved questions which we previously described.

V.A. Thrust I. Some Experiments in Building Prototype Software Support Systems

1. Studies of Existing Successful Support Systems

We believe that a wide variety of useful insights can be gained by close examination of the slowly growing number of successful

support systems currently in use. INTERLISP [Teit 78] and EL1 [ECL 74], already mentioned, are two examples of extant successful development support systems. The UNIX (TM) operating system [Ritc 74], along with its elaborate set of existing coordinated tools, seems to be a good basis for the construction of a variety of other support systems.

It would be quite profitable to study these examples in an attempt to determine what makes them successful. It would be most useful, for example, to study their user interfaces to see which characteristics seem to make interfaces popular and useful. It would be helpful to determine, for example, what levels of HELP (tutorial) systems, graphics support, and directory assistance seem minimally necessary to insure utility and popularity in these systems. By studying a variety of support systems, we should furthermore be able to gain insight into which features are generally useful, and which are perhaps desirable only for limited classes of users.

Another important type of understanding, obtainable by such studies of examples, is an understanding of the importance of tight coupling of tools. It is maintained that the popularity of INTERLISP and EL1 derive directly and inherently from the specialization of their support tools to a single subject language. UNIX-based support systems, on the other hand, are constructed from capabilities which, in general, themselves have no special language knowledge. Study of these examples can and must help us to come to an understanding of the extent to which language knowledge in tools is important. Study of examples and basic tool building research must then enable us to formulate notions of how language-intelligent tools and tool fragments can be efficiently created

from a base of more general tools and techniques.

Study of these existing systems should also help us to better understand the data base/information management system requirements for environments. Each of these existing support systems seems to maintain, to some degree of rigor and formality, a central information base. The success of the various strategies in meeting user needs can surely be studied. In particular, it would be important to study the type and amount of information retained, and the acceptability of these retainment policies to users.

Each support system has also adopted, apparently only implicitly, some model of its users and their activities. It would be interesting to formalize those models. From such formalizations could, for example, be determined the breadth of support extended by each system. User surveys could help determine the uniformity and strength of such support. This would help determine the ranges of applicability which are feasible for current support systems.

The models would also enable a determination of the flexibility currently offered by such systems. It has been hypothesized that environments must accommodate themselves to their users' way of getting their jobs done, not vice versa. It is important to determine whether current support systems do this. If not, it is important to determine whether this lack of flexibility is a significant source of dissatisfaction. It would also be important to decide if any such lack of flexibility is an essential consequence of tight integration, or whether perhaps better tool fragments could be used to achieve both tight integration and flexibility.

The list of things which should be studied in existing support

systems could go on indefinitely. Perhaps it is best to close simply by observing that much experimental work can be carried out on currently existing support systems. This work should be used to guide the creation of new experimental systems in the direction of systems which can provide elaborative, rather than duplicative, insights and understanding.

2. Tool Fragment Studies

This would be an experimental program aimed at identifying useful sets of tool fragments, and the extent to which they can support tight integration in the desired tool capabilities.

The experimentation would begin with the designation and assembly of a reasonable set of fragments to meet the needs of a specific, sharply circumscribed software activity. For example lexical analysis, parsing, flowgraph generation, verification condition generation, theorem proving, data flow analysis, and test probe insertion fragments might be designated as the fragments necessary to provide total comfortable support for the verification and testing activity. An actual verification and testing procedure would then be hypothesized in formal detail. The selected tool fragments would be configured and integrated to support this activity. The experimental program would be continued by attempting to reconfigure the tool fragments in response to a variety of changes, such as changes in the testing and verification procedures to be supported, change in the source language, and change in the user community (e.g., the addition of managers as observers of the activity).

In the process of adapting to these changes, the tool fragments will be altered to provide needed flexibility, or perhaps the need for

new or different sets of tool fragments will be recognized. In an important sense, this line of study is aimed at starting to understand what, if any, analogy exists between software production and the manufacturing concepts of interchangeable parts and assembly line production. As in manufacturing, it is expected we will discover that the utility of these notions is not uniform. That is to say, we expect to find that useful sets of tool fragments can be successfully produced to form the basis for construction of some types of software environments, but perhaps not all. Thus experimentation with a wide variety of tool fragment sets seems indicated.

3. Data Base Studies

This would be an experimental program aimed at determining ways in which informational bases for environments can be adequately stored, accessed and maintained. Following our conjecture that data bases for environments should be structured in accordance with models of the software jobs supported, this research project would begin by creating precise detailed models of various software jobs. This activity would be useful in itself as such models are extremely rare or nonexistent. From these models, data base schema would be designed and information management systems either built, adapted or simulated. The important experimentation would then entail the actual or simulated use of these data base/information management systems to determine the performance requirements on them.

What must be determined are the sorts of demands which typical usage places on support system data bases and information management systems. Hence, for example, at first simulated streams of user requests should be directed towards the support system data base/

information management systems. (With the passage of time other research activities, such as the ones described previously, should result in the creation and instrumentation of actual support system prototypes. These might be used to capture actual streams of user requests.) Measurement probes, inserted in the information management systems could then determine the searching, updating and deleting operations implied by these requests, and the required rates at which these operations must be performed. These measurements are needed in order to enable data base implementors to design schema which will facilitate efficient data base operations in support of expected user request sequences.

This sort of experimentation should help provide answers to the questions of how much information the data base should store. Clearly information storage becomes excessive when it significantly hinders frequently occurring searching and/or updating operations. Actual measurements taken on a variety of simulated support systems should elucidate these issues. Presumably different types of support systems will be found to be amenable to the maintenance of different amounts of archival storage. This experimentation might also suggest useful hierarchical storage schemes.

From our discoveries about what constitutes excessively large data bases should flow a variety of useful derivative information. We should find out, for example, whether very broad scope environments do or do not place excessive demands on the data bases and information management systems that must support them. We should find some environments in which the rate of updating is sufficiently low to make the storage of redundant and derived information acceptable or even

profitable. We should also be able to identify and perhaps characterize environments in which the storing of redundant information is impractical because of frequently occurring changes.

It is expected that this line of research might well serve as a stimulus for data base and information management systems research, as we suspect that currently available technology will be found to be unsatisfactory in meeting the needs of environment data bases and information management systems.

V.B Thrust II: Towards Creation of General-Purpose Software Environments

1. Construction of a General Purpose Environment to a Given Set of Specifications

The purpose of this activity would be to actually construct a general-purpose software environment. It is assumed that this activity will not commence until after at least three years' experimental implementation, aimed at least partially, towards this goal. Hence this effort will build upon a base of successes in significant tool integration, in understanding the ingredients of a viable user interface, and in identifying software activities well enough understood and supported to be subjects for broad, strong, uniform support by tools. The totality of such experience should place active workers in the field in a position to clearly specify a general area of software activity and how it is to be supported by a software environment. An example of what seems achievable in this period might be an environment to support the needs of a medium sized team in at least coding and design of batch processing software written in a small set of closely related language dialects.

It should be stressed that this goal is not likely to be satisfactorily achieved without an experimental learning process such as

was previously described. Starting from such a knowledge base, however, the goal of successfully producing environments of this scope seems reasonable. A program of continued experimentation and development should continue to broaden the scope of the environments produced, within bounds which should become more clearly understood as our ambitions grow.

2. An Experimental Test Bed for Configuring Environments

A far more ambitious and wide-ranging activity would be the assembling of a very wide assortment of tools and tool fragments with the goal of trying to configure them into a variety of environments. This activity would differ essentially from the previously described activity. It would not be directed towards the creation of single environments based upon the best products of earlier experimentation. Instead it would be directed at determining the range of environments producible from a fixed set of capabilities, and at evaluating the strengths and weaknesses of competing tools and tool fragments in the overall context of usage in a general environment. In a sense this is tantamount to a laboratory for environmental experimentation, and the activity supported a forerunner of environmental engineering.

This sort of endeavor seems to entail greater risk, as we currently have little experience in integrating software tools on so large a scale. Thus we currently have no basis for believing that such large sets of tools can be readily reconfigured to allow for the rapid construction of alternative environments which would be necessary for comparative evaluation. Perhaps the most realistic appraisal of this activity is that it is destined, at least within the next five years to be done by a free marketplace, rather than a central facility.

IV. Acknowledgments

This paper summarizes the discussions and conclusions of one of the four groups at the Workshop on Programming Environments held April 30 - May 2, 1980 at Rancho Santa Fe, California. The Workshop was sponsored by the National Bureau of Standards' Institute for Computer Science and Technology and was organized by Dr. Martha Branstad of NBS and Dr. W. Richards Adrion, now of the National Science Foundation. My special thanks go to them for making this stimulating workshop possible. I also wish to thank the other members of discussion group 2:

Lori Clarke, University of Massachusetts
Donald Good, University of Texas
Raymond Houghton, National Bureau of Standards
Thomas Love, International Telephone and Telegraph
Patricia Santoni, U.S. Naval Ocean Systems Center
Daniel Teichroew, University of Michigan
Anthony Wasserman, University of California at San Francisco

for their willingness to share their thoughts and experiences.

References

[Bell 77] T. E. Bell, D. C. Bixler and M. E. Dyer, "An Extendable Approach to Computer-Aided Software Requirements Engineering," IEEE Transactions on Software Engineering, SE-3, pp. 49-60, (January 1977).

[ECL 74] "ECL Programmers Manual," Center for Research in Computing Technology, Harvard Univ., TR 23-74, 1974.

[Ploe 79] E. Ploedereder, "Symbolic Evaluation of User-Defined Procedures in EL1," Center for Research in Computing Technology, Harvard Univ., TR 01-79, 1979.

[Ritc 74] D. M. Ritchie and K. Thompson, "The UNIX Time-Sharing System," CACM, 17, pp. 365-375, (July 1974).

[Step 78] S. A. Stephens and L. L. Tripp, "Requirements Expression and Verification Aid," Proceedings Third International Conference on Software Engineering, IEEE Cat. #78CH1317-7C.

[Ross 77] D. T. Ross and K. E. Schoman, Jr., "Structured Analysis for Requirement Definition," IEEE Transactions on Software Engineering, SE-3, pp. 6-15, (January 1977).

[Teit 78] W. Teitelman, et al. Interlisp Reference Manual, Xerox Palo Alto Research Center, September 1978.