Software Productivity Tools

Digital Technical Journal

Digital Equipment Corporation

Number 6
February 1988

The *Digital Technical Journal* is published by Digital Equipment Corporation, 77 Reed Road, Hudson, Massachusetts 01749.

Changes of address should be sent to Digital Equipment Corporation, attention: List Maintenance, 10 Forbes Road, Northboro, MA 01532. Please include the address label with changes marked.

Comments on the content of any paper are welcomed. Write to the editor at Mail Stop HLO2-3/K11 at the published-by address. Comments can also be sent on the ENET to RDVAX::BLAKE or on the ARPANET to BLAKE%RDVAX.DEC@DECWRL.

ISBN 1-55558-005-X

Documentation Number EY-8259E-DP

Book production was done by Digital's Educational Services Media Communications Group in Bedford, MA.

Cover Design

The helix of the DNA molecule depicted on our cover is a visual metaphor for software productivity tools, the theme of this issue. Just as the encoded DNA molecule serves as a template for the synthesis of new forms, so software productivity languages and procedures serve as tools for the development of new software programs. The image was created using the Lightspeed System.

The cover was designed by Barbara Grzeslo and David Carroll of the Graphic Design Department.

Contents

Editor's Introduction

Jane C. Blake
Editor

This issue of the *Digital Technical Journal* features papers on software productivity tools that assist programmers in the development of high-quality, reliable software. In addition to papers about these tools, we also present several papers that examine innovative project practices developed by Digital engineers to improve productivity.

Our first paper looks at the set of tools developed to support all stages of the software life cycle, from the requirements and specification stages through the maintenance stage. Bert Beander gives an overview of each of the tools and describes how their strong integration provides for a rich development environment.

Our second paper is not about a software tool, but rather about a study to determine to what degree software tools and new development methods are contributing to reductions in project cost and to increases in product quality. Anne Smith Duncan and Tom Harris discuss the influences on software productivity and present findings for three productivity metrics.

The subject of the next paper is the VAX Language-Sensitive Editor, an important component of the VAX/VMS software development environment. Glenn Lupton reviews the research on which the requirements for this advanced text editor were founded and then describes the design of various LSE features.

The next two papers are about languages that have been integrated with the VMS environment and provide programmers increased efficiency in the creation of programs. First, Steve Greenwood describes the VAX SCAN product and gives examples of how this rule-based text processing language simplifies the building of software, thereby reducing program development time. Next, Bob Conti presents an informative discussion of the inherent productivity features of the Ada language and the additional features provided by Digital's implementation.

High-level, functional interfaces for graphics programming, specifically the VAX GKS and VAX PHIGS implementations, are the topics of our next paper. Brian Axtell, Bill Clifford, and Jeff Saltz relate how these interfaces have made graphics programming easier and describe the common architecture on which both products are based.

The designer of a software tool is sometimes faced with the dilemma of choosing between flexibility and ease of use. Lew Lasher discusses how the designers of VAX RALLY, a forms-based fourth-generation language, resolved this issue through the design of RALLY's application definition system and run-time environment.

Also designed for application development, the VAX VTX and VAX VALU tool set allows the development and integration of applicatons in distributed, heterogeneous environments. Linda Benson, Mike Gianatassio, and Karen McKeen describe the VTX and VALU features and how these serve to enhance productivity.

The next four papers offer insights into some of the tools and techniques used by Digital software engineers to reduce project development time. In the first paper, Ron Brender, Bevin Brett, and Charlie Mitchell describe how their use of automation, instrumentation, self-checking, and self-description not only saved development time but also contributed to the VAX Ada compiler's performance. Next, Steve Grass discusses a new approach devised to manage the development of a then unprecedented graphical interface, the VAX COBOL GENERATOR software. In the third paper, Linda Ziman and Martin Dickau attribute significant time savings and product improvements to an iterative approach and the software tools used to develop the VAX DEC/Test Manager software. One of these tools was the VAX NOTES computer conferencing system, which is the topic of the fourth paper. Peter Gilbert reviews the innovative design and development strategies that led to the success of NOTES and describes several key product features.

In our final paper, Michael Good discusses the three principal activities of software usability engineering. He also gives examples of how this user-oriented approach has contributed to software product design at Digital.

We thank John Henning for his help in preparing this issue.

Jane Blake

Biographies

Brian A. Axtell A principal software engineer in the Core Applications Group, Brian Axtell is the project leader and supervisor for the VAX PHIGS product. Joining Digital in 1980, Brian co-designed the Base Graphics Architecture, the VAX PHIGS and VAX GKS products, and was also the project leader in the development of VAX GKS. He earned a B.S. in meteorology (1978) and a B.S. in computer science (1980) from the Pennsylvania State University. Brian is a member of the ACM, the SIGGRAPH Special Interest Group, and the American Meteorological Society.

Bert Beander Bert Beander, a consulting software engineer, joined Digital after receiving his Ph.D. and M.S. degrees in computer sciences from the University of Wisconsin in Madison. Prior to his current work in the area of programming environments with the Technical Languages and Environments Group, he supervised the VAX Debugger and the VAX Performance and Coverage Analyzer (PCA) projects and also served as project leader in the development of PCA version 1 and the Debugger versions 3 and 4. Bert is a member of the ACM, SIGPLAN, and SIGSOFT.

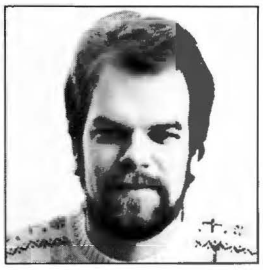
Linda E. Benson Linda Benson, a principal engineer, is the project leader and supervisor for the VAX VTX and VAX VALU products. Previously, she was development engineer and project leader for the VAX Run-time Library and has also been involved in a project to produce terminal front-ends. Prior to joining Digital in 1979, Linda was a software engineer at Grumman Aerospace Corporation where she contributed to the development of communication and navigation software systems. She received her B.S.C.S. in 1977 from Rochester Institute of Technology, Rochester, N.Y., and is currently working toward an M.B.A. degree.

Ronald F. Brender As a senior consultant software engineer, Ron is responsible for most static semantics processing in the VAX Ada compiler. Prior to this work, he supervised development of the first optimizing compiler on the PDP-11 and the development of the current generation of BLISS language and implementations. He has been appointed by the Ada Joint Program Office to the Ada Board and was a member of X3J3, the ANSI committee that developed the current FORTRAN '77 standard. Ron received his B.S.E. in engineering science, M.S. in applied mathematics, and Ph.D. in computer and communication sciences from the University of Michigan.

Bevin R. Brett Bevin Brett is a principal software engineer in the Technical Languages and Environments Group, currently working on the VAX Ada project. Most recently, he has been involved in the Integrated Programming Support Environment task force and in the design and implementation of shared generic instantiations in VAX Ada compiler code. Before joining Digital in 1982, Bevin received an M.Sc. in computer science (1982) from the University of Adelaide, a B.Sc. (Honors, 1977) in mathematics from the University of Canterbury, and graduated as valedictorian from the Nelson Boys College (1974), New Zealand.

William H. Clifford Jr. Bill Clifford, a principal software engineer, is a co-designer of Digital's Base Graphics Architecture and the VAX PHIGS product, and of the proposed three-dimensional extension of X11. Bill is a representative from Digital to the ANSI X3H3.1 (PHIGS) committee and to the ad hoc PHIGS+ committee. Before joining Digital in 1984, he developed real-time, distributed control systems at Stable Technology, Inc. He received a B.S. (1968) and an M.S. (1970) in systems engineering from Case Western Reserve University and pursued doctoral work as an NDEA Fellow at the university.

Robert A. Conti A consulting software engineer, Bob Conti has contributed to the VAX Ada project in the area of tasking and debugging support. He is currently pursuing advanced development work related to future enhancements to VAX Ada. Before joining Digital in 1981, Bob developed software for several radar systems, including AWACS, at Westinghouse. He received a B.S. in engineering (1968) from Case Western Reserve University, an M.S.E.E. (1971) from Johns Hopkins, and an M.S.C.S (1981) from the University of Maryland. Bob is a member of Tau Beta Pi, Eta Kappa Nu, IEEE, and ACM.



Martin Dickau Martin Dickau is a Senior Software Engineer in the Commercial Languages and Tools development group. Currently a developer with the VAX Software Project Manager project, he worked on the VAX DEC/Test Manager versions 2.0 and 2.1 and was acting project leader on the DEC/CMS project. Martin was a coop student at Digital for two years and joined the company after receiving a B.S. in computer science from the Massachusetts Institute of Technology in 1985.

Anne Smith Duncan Anne Smith Duncan, a software engineering manager in the Commercial Languages and Tools Development Group, is working in the area of measuring and improving the software development process. She has also held various technical and managerial positions in the Distributed Information Systems Group and the Software Standards Group. Prior to joining Digital in 1978, Anne worked at the Department of the Navy as a Computer Specialist in application systems development and technical support. She earned a Certificate of Management from the Smith College Management Program (1984) and a B.A. from George Washington University (1969).

Michael Gianatassio Jr. Mike Gianatassio joined the VTX and VALU engineering team in 1983 and with that team has had responsibilities in the areas of development, consulting, and product architecture. As part of the Videotex Technical Partnership program, he worked with financial institutions on the design of home banking systems. Mike is currently a principal engineer and project leader in the IBM Interconnect Group. Prior to joining Digital in 1982, he worked at Polaroid Corporation. He earned a B.S. degree (1982) in computer science from Northeastern University of Boston.

Peter D Gilbert After earning a B.S. in computer science (1976) and an M.S. in computer engineering (1979) from the University of Illinois, Peter Gilbert joined Digital in January 1979. He is a member of the Commercial Languages and Tools Group and has been a developer on the VAX COBOL, VAX and PDP-11 sort/merge, and VAX NOTES projects, and was a developer responsible for the collating sequences, parallel processing, and mathematics software for the VAX Run-time Library project. Currently working on the design of a configuration management tool, Peter is a principal software engineer.

Michael D. Good As a principal software engineer in the Software Usability Engineering group, Michael Good is developing software usability engineering methodologies and contributing to the user-interface design of several products. He has conducted usability research since joining Digital in 1981 and has published a number of papers on usability engineering and text editing. He designed and implemented the Eve text editor for the VAX/VMS operating system version 4.2. Michael received a B.S. (1979) and an M.S. (1981) in computer science from the Massachusetts Institute of Technology.

Steven J. Grass Steve Grass, a principal software engineer in the Commercial Languages and Tools Group, is the project leader of a group responsible for the development of common components for window-based applications. Previously, he worked on the VAX COBOL GENERATOR project, first as a member of the advanced development team and then as the project leader for implementations of versions 1.0 and 1.1. Steve was also a developer on the PDP-11 COBOL and COBOL-81 compiler projects. He joined Digital in 1978 after earning a B.S.E. in computer engineering from the University of Michigan.

Stephen R. Greenwood A consulting software engineer, Steve Greenwood is currently responsible for a tool for specifying the end-user interface to window-based applications. Previously, he has been involved with the design of run-time libraries for future architectures and the design and development of the VAX SCAN language. Before joining Digital in 1981, Steve worked at Sperry Univac Corporation as a member of the compiler development group. He received his B.S. in physics (cum laude, 1973) from Bucknell University and an M.S. in computer science (1975) from the University of Wisconsin. He is a member of the ACM.

Thomas J. Harris Since joining Digital in 1978, Tom Harris has been a manager of the Commercial Languages and Tools Development Group and is currently the group's senior manager. He chairs Digital's Sponsored Research Board and is responsible for advanced development planning across the Systems Software Development Group. Prior to joining Digital, he worked for Sperry Univac Corporation in software and hardware product development. Tom is a member of the CODASYL Executive Committee. He participated in and chaired the CODASYL Command Language Committee and was a member of the FORTRAN Data Manipulation Committee. Tom earned a B.S. in engineering at Case Western Reserve University in 1967.

Lewis Lasher Lew Lasher is a senior software engineer in the Software Development Technologies Group. Since joining Digital in 1985, he has worked on the VAX RALLY project, primarily in the area of user interface. He earned an A.B. degree in applied mathematics in 1978 at Harvard College, where he served as a teaching fellow in undergraduate courses in computer science. While at Harvard he also worked on PPL, an interpreted language used there in the introductory programming course. Lew earned a J.D. degree at Harvard in 1981 before returning to software engineering.

Glenn H. Lupton Glenn Lupton joined Digital in 1975 after receiving a B.S.E.E. (1973) and an M.E.E.E. (1974) from Rensselaer Polytechnic Institute. A consulting software engineer in the Technical Languages and Environments Group, he is the project leader of the VAX Language-Sensitive Editor project. Glenn has been associated with the BLISS compiler projects as developer, project leader, and supervisor. He has also supervised the development of a number of software programming environment tools, including the prototypes for VAX SCA, DEC/MMS and the DEC/Test Manager software.

Karen L. McKeen Karen McKeen is a senior software engineer working in the VAX VTX/VALU engineering group. She joined the VTX group in 1985 and has been responsible for designing and developing information provider components. She is currently the architect for VTX and VALU. Previously, Karen was project leader for the VAX DECgraph project. She has also worked in the Commercial Systems Evaluation Group developing performance tools. Karen joined Digital in 1979 after earning a B.S in mathematics with a computer science interdisciplinary option from the University of New Hampshire.

Charles Z. Mitchell Charlie Mitchell, a consulting software engineer in the Technical Languages and Environments Group, has been a member of the VAX Ada development project since its inception as an advanced development project in 1979. Currently the project leader of the VAX Ada development project, he joined Digital in 1976 as a developer in the LCG Languages Group. Charlie received a B.S. from the University of New Mexico and an M.S. in computer science from Rensselaer Polytechnic Institute. He is a member of the ACM and SIGAda.

Jeffrey S. Saltz Jeff Saltz joined Digital after receiving a B.S. in computer science (honors, 1985) from Cornell University. A senior software engineer in the Core Applications Group, he is co-designer of Digital's Base Graphics Architecture, and the VAX PHIGS and VAX GKS products. Jeff is a representative from Digital to the ad hoc committee for the proposed three-dimensional extension to X11 and a co-architect of the X3D-PEX proposal. He is a member of Tau Beta Phi.

Linda Ziman Linda Ziman is a development supervisor in the Commercial Languages and Tools Group. She is currently responsible for several projects, including the VAX DEC/Test Manager, VAX DEC/CMS, VAX Software Project Manager, and an advanced development project. Previously, Linda worked with integrated software environments and was project leader of the DEC/Test Manager project. She has worked in the area of software productivity tools since joining Digital in 1978. She received her B.S. degree from Union College and is a member of ACM and IEEE.

Foreword

William J. Heffner
Vice President,
Systems Software Group

What is a programmer? What does he/she do? Why does it take so long? These are three of the questions most often asked of those of us in the software profession.

Augusta Ada Byron (1815–1852), the Countess of Lovelace, has been accorded the title of the world's first programmer. Her notes published in London in 1843 regarding Charles Babbage's analytical engine included a formula for solving a problem on that machine. This formula is in effect the first example of a computer program, hence her recognition as a programmer.

For roughly the century following Byron's notes, the person who designed the computer also built and used the computer. There was seldom a separation of the builder and the user. In the middle of this century, however, computers were being used by many people not involved in either designing or building the computer. These users transformed their problem statement into a computational method understood by the computer. This computer representation of the problem was called a program, and the person preparing it was called a programmer.

Now, what is a programmer? Simply put, the programmer is someone heretofore unknown in the professions. Programmers in the 1950s and 1960s came from many disciplines. Many were electrical engineers and mathematicians, but others were musicians, liberal arts majors, and even dentists, hospital administrators and the like. What was the unique talent they possessed? In his book *The Mythical Man Month*, Fred Brooks likens a programmer to a poet in that a creative, intangible product is the result of a programmer's work. Even though colleges and universities have formalized the training of programmers in a discipline called software engineering, we are certain only that programmers write programs; that the discipline is unique; and that because this discipline is unique, programmers require unique tools and products to effectively complete their tasks.

The papers in this issue of the *Digital Technical Journal* address a part of our continual effort at Digital to produce the environment and products that assist programmers in producing timely, well-defined, efficient, and reliable programs. Historically, there have been two major breakthroughs in reducing the elapsed time to produce a working program. First were compilers, which provide the programmer a more concise and error-free technique for producing programs. Grace Hopper at UNIVAC and John Backus at IBM were leaders in this breakthrough. The second major breakthrough, led by Digital, was interactive timesharing. Interactive timesharing allowed the programmer greater access to the computer, thus reducing the elapsed time for program development.

In addition to the effort to reduce elapsed time, equal effort is being expended to add discipline and predictability to the process of producing programs. Today, very few programs are developed by a single programmer. Instead, teams of programmers collaborate to produce larger and more comprehensive programs, for example, the FORTRAN project and the VMS project. To accomplish such projects, programmer productivity tools and the Computer Aided Software Environment (CASE) have been developed. The VAX/VMS system has been the industry standard for programming development and the system of choice for programmers. The papers herein demonstrate our continuing effort to be the leader. Our goal is to produce the best environment for programmers — an environment in which they can exploit their creativity as they participate in a predictable and disciplined process.

Bert Beander

VAX/VMS Software Development Environment

The VAX/VMS software development environment comprises tools that support all stages of the software life cycle. These tools include documentation tools, a project management tool, code management and system building facilities, a rich editing and browsing environment, a powerful debugger, static and dynamic analysis tools, test management facilities, and project communications tools. Moreover, these tools are strongly integrated with each other: they share a common user interface philosophy, they have numerous tool-to-tool links that allow them to pass substantial amounts of program information to each other, and they support multiple programming languages. As a result, the environment has both richness and internal cohesiveness.

Software development has become increasingly dependent on programming environments that provide a rich set of software development tools. Such environments are attractive because they can increase both programmer productivity and software quality, while reducing development costs. The programming environment that Digital has developed for the VAX/VMS operating system is an example of a commercially available environment that provides a particularly rich set of tools.[1] This environment has evolved from the handful of compilers and tools that were available when the original VAX-11/780 system was introduced in 1978. A majority of the tools, however, have been developed since the early 1980s.

As a result of this development, the VAX/VMS programming environment now provides a set of tools that satisfies two goals. One is that the tools should assist the software developer in all stages of the software life cycle. All stages have tasks that can be automated for greater programmer productivity, and no stage should become the principal bottleneck in the development process. The other goal is that the tools should work well together so that they provide an easy-to-use, consistent, and tightly integrated environment for the user. Tight integration increases programmer productivity because program data collected by one tool can help automate the functions of other tools, and consistency between tools increases productivity by reducing developer learning time. This paper describes how the separate tools of the VAX/VMS software development environment support the various stages of the software life cycle and explains how the many tool-to-tool links and information flows make the environment so tightly integrated.

Supporting the Software Life Cycle

Digital's programming environment on the VAX/VMS operating system provides a rich set of tools designed to support all stages of the software life cycle. The software life cycle includes the following stages:

- The requirements and specification stages, when documents are written to define the software project
- The design stage, when data structures and program components are designed
- The implementation stage, when code is written, debugged, and tested
- The testing stage, when new software is tested by users
- The maintenance stage, when bugs are fixed and minor enhancements are added

At each of these stages, software developers use tools that are specific to that stage. In addition, they use certain tools in all stages of the life cycle to maintain project artifacts, such as

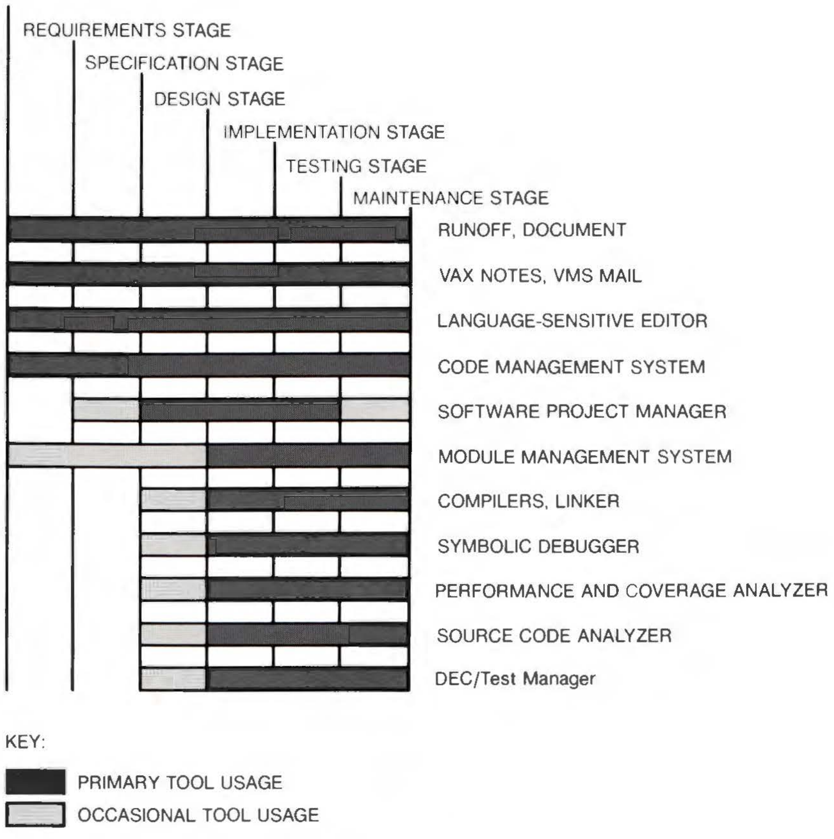


Figure 1 The Software Life Cycle

documents and source files, and to manage project activities. This section describes the stages of the software life cycle and the tools that are used at each stage.

Figure 1 summarizes the software development stages and the associated tools. In this diagram, the life-cycle stages are listed along the top and selected tools are listed along the right side. Solid bars mark the life-cycle stages where tools have their primary uses; light bars mark the stages where tools are occasionally used.

Requirements and Specification Stages

During the requirements stage, the customers or developers identify the requirements of the proposed software system. During the specification stage that follows, developers formulate detailed specifications that define what the system will do and how it will be used. By comparing the specifications to the requirements, the developers can show, at least informally, that if the system is built as specified, it will meet its requirements.

Both requirements and specifications are usually written in English or another natural language. The tools needed at these two stages must thus facilitate the production and organization of documents. To produce documents, developers first need one of the environment's text editors, such as the VAX Text Processing Utility or the VAX Language-Sensitive Editor (described further below), to compose the actual text. They then need a text processing tool to format their documents. Two such tools are available on VAX/VMS. One is Runoff, which produces documents as formatted ASCII text files. Runoff is simple but quite serviceable for documents that only require typewriter quality. VAX DOCUMENT is a newer tool, which has been used at Digital to produce all VAX/VMS software documentation. DOCUMENT converts text files written in a markup language into typeset-quality, formatted output. The output can be printed on a laser printer or processed on a typesetting system for final production. DOCUMENT is layered on top of Donald Knuth's TeX text processing system, and thus supports multiple fonts, mathematical typesetting, and extensive formatting capabilities.[2]

Documents also need to be stored. Although they can be stored as ordinary files in ordinary VMS directories, it often makes more sense to use the VAX DEC/CMS (Code Management System) tool to store documents. CMS can store multiple versions of document sources efficiently, and it allows old versions to be retrieved at any time. CMS also provides check-out/check-in control over document sources to prevent different developers from inadvertently modifying the same sources at the same time. A developer thus checks out (or "reserves") a source file from a CMS library into a private work area, works on it in the private area until satisfied with it, and then checks it back in (or "replaces" it) to the CMS library. While the source module is reserved, no other developer can modify it. (CMS allows multiple concurrent reservations, but developers who choose this option must be willing to later merge the independently made changes. CMS has facilities that partially automate such merges.)

Another tool that is very useful when collecting requirements is the VAX NOTES electronic conferencing system.[3] NOTES allows multiple users to share comments on a variety of topics. Each NOTES conference is organized into "topics," where a written note starts the discussion of each topic. Members of the conference can create new topics at any time, and they can reply to existing notes and other people's replies. All information is stored on line and is easily perused from any node in the users' computer network. VAX NOTES thus provides a very convenient and expedient way to collect requirements and reviewers' comments for a software project, and is widely used within Digital during the requirements and specification stages of the software life cycle. The VMS Mail utility is also used extensively for project communications and for information exchange with other groups.

Design Stage

During the design stage, developers design the data structures and program components that will constitute the implementation of the proposed software system. Developers usually write documents to describe their designs; but in addition, they normally define selected data structures and routine headers at this stage. These components are written in programming languages and are thus created using editors. The VAX Language-Sensitive Editor (LSE) is usually the editor of choice.[4] In this section, we discuss how LSE is used and also mention how designs may be represented graphically. Once a design is in place, the developers must formulate a plan for building the desired software and create a development schedule based on that plan. A discussion of a tool that helps developers do such planning concludes this section.

The design components written in programming languages are normally created using LSE. LSE is a full-featured, programmable, full-screen text editor. It is "language-sensitive" in several senses. First, it provides templates for the constructs in each supported programming language (about a dozen languages are currently supported and users can create templates for additional languages). Second, it allows placeholders in those templates to be expanded so that the valid possibilities for each syntactic entity can be displayed and selected. Third, it provides on-line help for each supported language. And fourth, it allows programs to be compiled directly from the editor and compilation errors to be reviewed directly in the editor.

These capabilities are best explained by example. Suppose a user wants to enter a `WHILE` loop in a Pascal program. To do so, the user enters the `WHILE` keyword and then expands that construct by pressing an "expand" key. In response, LSE produces the following text:

```
WHILE %{expression}% DO
    %{statement}%
```

Within this template, there are two placeholders: one for the Boolean expression, and one for the statement that forms the loop body. Single keystrokes move the editing cursor from placeholder to placeholder. Any placeholder can in turn be expanded to display a list of valid alternative expansions from which the user can choose one. For example, pressing the expand key when the cursor is on the `%{statement}%` placeholder displays a list of valid Pascal statement types. The user can then choose the desired statement type and expand it to get its template inserted into the text buffer. Alternatively, the user can simply type over the placeholder to replace it with the desired program text.

The expansion of placeholders into templates is by itself a powerful form of language help because it enables programmers to produce syntactically correct programs even if they do not know the language very well. In addition, LSE provides language help in the form of help text that explains the form and usage of each language construct.

Finally, developers can compile programs from LSE and review compilation errors in the editor. To compile a program, LSE writes the contents of the current buffer to a file, creates a subprocess, and runs the compiler on that file in the subprocess. The compiler records any error messages in a "diagnostics file," which it passes back to the editor. The editor displays these error messages in one editing window while displaying source code in another window. The user can select successive error messages and direct the editor to automatically position the source window on the corresponding error locations. Errors are thus quickly located and corrected. Some compilers will also suggest error corrections, in which case LSE automatically displays those corrections in the source window for the user's approval or disapproval.

When designing data structures, developers may choose to store their data definitions in the VAX CDD (Common Data Dictionary) database. CDD serves as a repository for data definitions common to many separate programs, where the programs access common databases and may be written in many different languages. CDD is particularly well suited to commercial environments where multiple applications programs access large central databases.

Developers may also create graphical representations of designs using techniques such as structured analysis, structured design, or data modeling. Digital does not itself provide tools to automate graphical software design, but suitable tools are available for the VMS operating system from other vendors such as Intech, Cadre, Nastec, Tektronix, and Interactive Development Environments.

Once a design is in place, the project leader must formulate a plan for building the desired software. To do so, he creates a work breakdown structure that identifies the individual tasks or work assignments needed to implement the design. He associates time estimates with the individual tasks, identifies dependencies between tasks, and determines which programmers are available. Given this information, the project leader then uses the VAX Software Project Manager (PM) tool to construct a project schedule that shows when each task will begin and end. By later recording the actual start and end dates of each task, the project leader can use PM to track actual progress and compare it to the schedule. The value of PM is that it automates much of the bookkeeping associated with scheduling and controlling a software project, thus helping to ensure that the project is completed on schedule. This kind of bookkeeping would otherwise have to be done manually.

Implementation Stage

During the implementation stage, code is written, debugged, and tested. The environment provides numerous tools for this stage. These tools include editors, compilers, a debugger, code management facilities, a system builder, and static and dynamic analysis tools. This section gives an overview of these tools.

When writing code, developers using the VAX/VMS software development environment can choose from among more than a dozen programming languages, and they may include modules written in different languages in the same program. The developers write most code using LSE, but may also use specialized editors such as a forms editor. Developers compile programs using both the standard language compilers and specialized compilers such as the message compiler (for error messages) and the help librarian (for creating hierarchical help text). They then link and run the program.

To debug their code, programmers use the VAX/VMS debugger.[5] The debugger allows the programmer to set breakpoints in the code, to set watchpoints (data breakpoints) on data locations, to single-step the program by source line or machine instruction, to examine variable values, and to deposit new values into memory, among many other things. The debugger is fully symbolic, receiving its symbol information from the compilers via the linker. The debugger uses multiple windows on the user's screen to display extensive program state information to the user. This information allows the user to find program bugs rapidly and efficiently.

To organize and maintain all program sources, the developers use the VAX DEC/CMS code management system, described earlier as a tool for managing document sources. To build the software system being developed, programmers use the VAX DEC/MMS (Module Management System) system builder. Like the UNIX Make utility, MMS performs a minimal system build based on module dependency information and knowledge of which source modules have changed since the last build.

To follow cross-references and perform static analysis, developers use the VAX Source Code Analyzer (SCA). SCA receives cross-reference information from the compilers. This information is incorporated into a database that allows cross-reference queries over an entire software project to be answered quickly. SCA is tightly integrated with LSE so that LSE can display cross-reference information and cross-referenced source code in editor windows. SCA can also perform static analysis by showing call trees and by checking procedure calls for consistency with the corresponding procedure declarations.

To perform dynamic program analysis, developers use the VAX Performance and Coverage Analyzer (PCA). PCA can collect several kinds of performance data during program execution, including program counter sampling data, page fault recording, I/O usage, and exact execution counts at specified program locations. PCA can later display all this data in a variety of histograms and tables. PCA can also show performance data at various resolutions, from the program module level down to the individual source line or even instruction. By using PCA, programmers can quickly locate performance bottlenecks, many of which usually turn out to be easy to remove by reprogramming. PCA thus helps programmers produce high-performance software, something that is hard to do without this kind of tool.

Testing Stage

There are typically two kinds of software testing. First, developers test the software during the implementation stage to ensure that all individual functions work. Second, actual users test the software to ensure that it works under normal operating conditions. Several components of the VAX/VMS software development environment were designed to help make programmers more productive by automating certain activities of the testing stage.

To test software during implementation, developers use the DEC/Test Manager (DTM) testing tool. To use DTM, developers must first write test scripts for their software, where each "script" consists of input to the software that will test various software functions. The developers then have DTM capture the software's output when the software is run under each script, and they manually certify that the software produces correct output for each script. DTM then saves the correct outputs as "benchmark files" and organizes the test scripts into user-defined categories. Subsequently, the developers can use DTM to automatically run various categories of tests (or all tests) on later versions of the software. When DTM runs a collection of tests, it runs a set of test scripts through the software being tested, collects the outputs from the software, compares the actual outputs to the expected outputs (the benchmark files), and reports any differences to the user. DTM allows developers to build up large regression test systems for their software. Experience indicates that such test systems constitute the single best guarantee of software quality.

The VAX Performance and Coverage Analyzer is important during testing because it can measure test coverage, that is, identify the code paths that are or are not executed by the regression tests. (PCA measures what some people would call "statement coverage"; PCA determines what instructions are executed, not what branches are taken.) The coverage is reported symbolically in source code displays. Using this information, developers can write additional test scripts to ensure that all code paths are tested at least once.

Once the software is implemented and passes all regression tests, it is ready to be tested by actual users in a field test. During field test, problems must be reported to the developers. Provided the users and the developers are on the same computer network, VAX NOTES has proven to be an excellent problem reporting tool. A user can report each new problem as a separate topic and developers can reply to each topic. Other users can see the problem reports along with their responses, which alerts them to known problems; they can also enter additional responses to supply further information or to answer questions.

Maintenance Stage

When a software system is released to its users, it enters the maintenance stage of the software life cycle. At this stage, bugs are fixed and minor enhancements are added. (Major enhancements require a new pass through the whole software life cycle, and developers start this process by defining the requirements for the next major version.) As during field test, NOTES can be an effective tool for recording and responding to problem reports, provided the users and developers are on the same computer network. As during implementation, the standard coding tools — LSE, the compilers, the linker, the debugger, and PCA — are used to fix bugs and add enhancements.

During the maintenance stage, CMS and MMS remain essential. CMS's ability to keep track of multiple versions of the software system and to maintain multiple parallel development streams (variants) of the program sources is particularly important. For example, by using CMS, developers can easily maintain a version 1.1 maintenance stream of the sources (for bug fixes) while also working on a version 2.0 development stream (for major enhancements). The Source Code Analyzer is also very useful because it makes it easy to browse through unfamiliar sources and quickly obtain the definitions of procedures, variables, and other program constructs.

Finally, the Test Manager remains very important at this stage for maintaining software quality as changes and bug fixes are made. A well-designed set of regression tests can ensure that all major functions of the software system still work correctly after changes have been made. Testing can never demonstrate the absence of errors, but the successful execution of well-designed tests can demonstrate that all common operations work correctly in typical circumstances. Such tests can therefore give developers a high degree of confidence in the integrity of the software.

Integration among Tools

To increase their usability and to enhance the smoothness with which they can be used together, Digital's tools are strongly integrated with each other. This integration takes three forms:

- All tools share a common command language philosophy. Consequently, commands have the same syntactic form and general appearance in all tools.
- A great deal of program information flows between tools. The compilers, in particular, generate a substantial amount of information for tools such as the debugger, the performance analyzer, the editor, and the static analysis and cross-reference tool. Other tools can invoke each other, passing along enough information to create a smooth transition from tool to tool.
- All tools support the development of applications written in multiple programming languages. Developers are therefore free to pick the language or languages they deem best for their applications.

The strong integration between tools gives the programming environment a mature, cohesive feel to the user. Because tools have been developed together, they can also give a wealth of capabilities which would not otherwise be possible. This section describes how the environment is integrated across tools and illustrates some of the capabilities that this integration makes possible.

Common Command Syntax

All tools in the VMS environment have command languages that are based on the same philosophy as the Digital Command Language (DCL), the top-level command language for VMS. In DCL, each command consists of a command name, followed by zero or more "qualifiers," followed in turn by zero or more command parameters. The following command, which invokes the FORTRAN compiler, is an example:

```
FORTRAN/DEBUG/NOOPT A,B
```

Here `FORTRAN` is the command keyword, `/DEBUG` and `/NOOPT` are qualifiers, and `A` and `B` are parameters. The command compiles files `A.FOR` and `B.FOR` with debugging information enabled and optimization disabled.

All tools in the VMS environment have commands of the same syntactic form as DCL. Furthermore, when tools have common capabilities, they use the same command syntax. For example, the `SPAWN` command, which creates a new subprocess, has the same syntax in DCL, the debugger, the Mail utility, the Language-Sensitive Editor, and many other tools. The help system also works the same way in all tools. As a result, all tools share a common "feel," and users can frequently guess how to use a given tool from their knowledge of other tools. Future workstation interfaces will maintain this uniformity across tools by having all tools use a new windowing interface conforming to the industry-wide X Window standard.

Information Flow between Tools

The integration of the VMS programming environment stems in large part from the information flow between tools. The compilers in particular generate a substantial amount of information for other tools. They generate symbol table information for the debugger and the Performance and Coverage Analyzer, diagnostic information for the Language-Sensitive Editor, and cross- reference and calling-sequence information for the Source Code Analyzer. The compilers are thus the sole sources of semantic program information, but they make that information available in suitable forms to all tools that need it. This section discusses these information flows and certain other connections between tools.

Figure 2 illustrates the many connections and information flows between tools that give the VAX/VMS programming environment its tight integration. The boxes represent tools, and the arrows represent information flows, either via files or through direct calls between tools.

The debug symbol table (DST) contains the name, type, and address or value of every symbol in the user's program. This information is passed from the compiler to the linker, which performs address relocation on the DST. The information is then passed to the debugger. The DST contains scope information so that the scope of each symbol is known to the debugger. The DST also contains the correlation between program counter

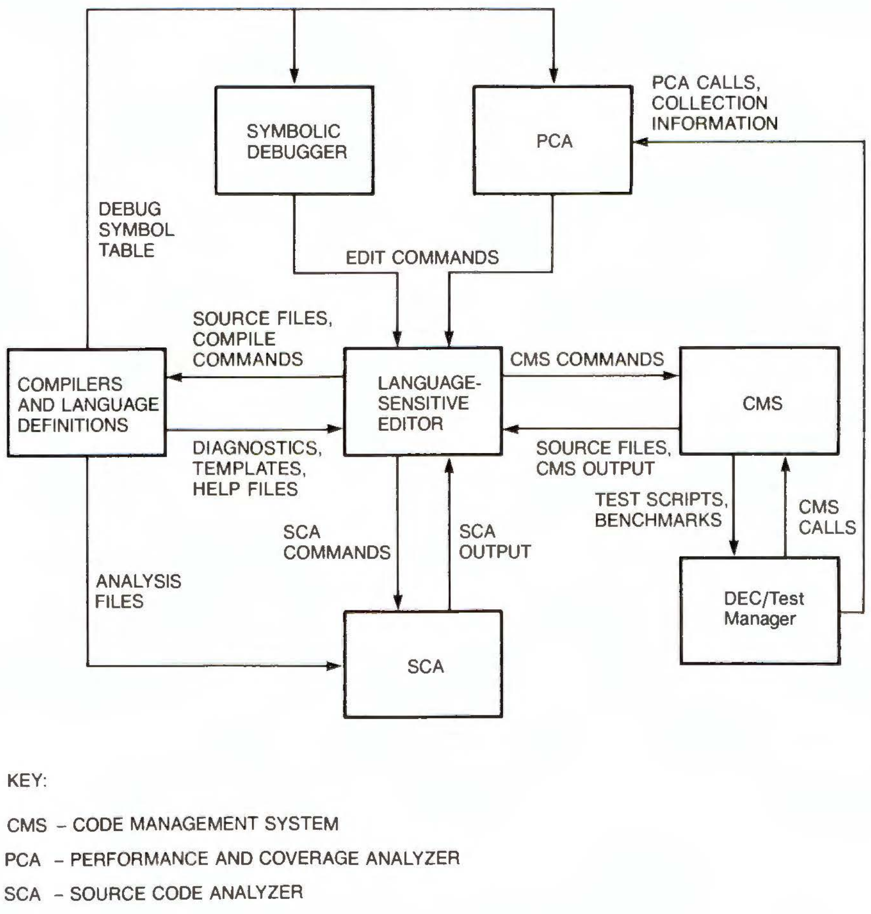

SYMBOLIC DEBUGGER
PCA
PCA CALLS, COLLECTION INFORMATION
DEBUG SYMBOL TABLE
EDIT COMMANDS
SOURCE FILES, COMPILE COMMANDS
COMPILERS AND LANGUAGE DEFINITIONS
LANGUAGE-SENSITIVE EDITOR
CMS COMMANDS
CMS
DIAGNOSTICS, TEMPLATES, HELP FILES
SOURCE FILES, CMS OUTPUT
TEST SCRIPTS, BENCHMARKS
CMS CALLS
SCA COMMANDS
SCA OUTPUT
ANALYSIS FILES
DEC/Test Manager
SCA
KEY:
CMS - CODE MANAGEMENT SYSTEM
PCA - PERFORMANCE AND COVERAGE ANALYZER
SCA - SOURCE CODE ANALYZER

Figure 2 Information Flows between Tools

values and source lines so that the debugger can display the source code that corresponds to specified run-time program addresses. PCA uses the same information to display performance and coverage data symbolically.

The diagnostic information is passed from compilers to the LSE editor via a diagnostics file, as described earlier. This information includes the text of each error message along with the source location of the error. If the compiler suggests error corrections, the suggested corrections are included too.

Language syntax (templates and placeholders) is passed to LSE through template files, and language-specific help is passed to LSE via help files. Although template and help files are not generated by the compilers as such, they are written by Digital's compiler developers. These files thus represent information flow from the compilers to LSE.

The compilers create "analysis files" to hold all cross-reference and static analysis information. These files can then be included in an SCA library, from which SCA can quickly answer cross-reference queries and perform call-tree and call-sequence analyses. The analysis file contains the name, type, and scope of each symbol in the user's program; its information thus partially overlaps the DST information. However, the analysis file also contains detailed information on all symbol references, including the type of each reference (read-reference, write-reference, declaration, etc.), and detailed calling sequence information on all procedure symbols.

LSE and SCA are separate tools that can be run separately. However, they are strongly integrated with each other so that any SCA command can be entered directly to LSE. Also, LSE win-dows can be used to display cross-reference and static analysis information, and cross-reference information can be used to automatically position editor windows at specific symbol references. This tight coupling between the two tools makes them look like a single tool to the user and gives the user a very rich editing and browsing environment for program sources. In fact, SCA is seldom used alone except in batch runs.

Other connections between tools pass more modest amounts of information, but still help provide a smooth, seamless feel to the environment. The debugger can display the source code corresponding to the current program location. If the user sees an error in that source code, he can enter the EDIT command, which causes the debugger to invoke LSE in a separate process and to pass the current source location to LSE. LSE positions the editing window to that source location, and the user can correct the source code immediately. PCA has the same connection to LSE. After editing the code, LSE can invoke the appropriate compiler, also in a separate process, and pass along the edited source.

If the user wishes to browse through sources stored in a CMS library, LSE is able to read those sources by calling CMS directly. There is also a RESERVE command in LSE which allows source modules to be checked out from a CMS library directly via the editor. Again, LSE calls CMS to do this. The Test Manager can also store test scripts and expected test results in a CMS library and will call CMS directly to retrieve those files.

A test run managed by the DEC/Test Manager is often a natural vehicle for collecting performance or coverage data. DTM therefore passes information to PCA (via VMS logical names) that tells PCA the name of each separate test script and the way the data should be collected. When the developer later uses DTM to review test results, he can invoke PCA directly from DTM to display the performance or coverage data associated with the current test execution.

In all these cases, Digital's tool developers have created connections between tools whenever they have been able to identify useful connections. Since most of these tools are developed in the same organization and most tool groups are physically close to each other, it is relatively easy for the developers of different tools to work together to develop the connections between tools that give the VAX/VMS programming environment its cohesiveness.

Multilanguage Support

One of the strengths of the VAX/VMS programming environment is its support of multiple programming languages. Software developers are thus free to choose the programming languages best suited to their applications, and they can include modules written in different languages in the same program. At present, the environment supports about a dozen languages. Only compiled languages are supported; interpreted languages have execution and editing models that do not readily fit into a compiled-language environment.

The programming environment supports multiple languages in two ways. First, all Digital compilers generate code that adheres to the VAX/VMS Calling Standard, which standardizes how programs call procedures and pass parameters. Because all compiled languages use this standard, modules written in different languages can always call each other, provided both languages understand the data types of the parameters.

Second, all the tools support multiple languages. The debugger can debug modules written in any language whose compiler passes symbol table information to it. LSE can support templates and placeholders for any language for which someone has constructed a template file, and it can review error messages from any compiler which passes diagnostics files to it. SCA can provide cross-reference services and static analysis for any language whose compiler creates analysis files. The Common Data Dictionary (CDD) can pass data definitions to any language whose compiler accepts them. To fully participate in the environment, each compiler must thus provide all the information needed by the various tools, and each compiler must call certain tools, such as CDD.

To support multiple languages, all tools use essentially the same implementation strategy. They define a single canonical representation for the data they need so that the same data from two different languages is always represented the same way. LSE has only one template file format and one diagnostics file format. All compilers describe a given data type or programming construct in the same way to the debugger. PCA uses the same symbol information as the debugger. SCA accepts only one format for its cross-reference information. If two languages pass a given piece of information to a given tool, they must always do it the same way.

However, all tools must also support the union of all constructs in all the programming languages they support. The debugger must support every data type that occurs in any language. It thus understands a numeric string type that occurs only in RPG (a report generation language) and tasking constructs that occur only in the Ada language. SCA must understand every kind of cross-reference and every kind of calling sequence that may come up in a multilanguage program, even though no one language has them all. PCA and the debugger must both understand case-sensitivity, which occurs only in C.

Multilanguage support thus complicates the design of most programming tools considerably. The tools must be designed to cope with a wide variety of language constructs. They must understand subtle semantic differences in apparently similar constructs in different languages. They must also be very extensible since it is impossible to predict what languages they may have to support in the future. As a result, the tools generally are very table-driven, and they are very dependent on having well-defined interfaces with the compilers and the other tools.

However, there are also substantial savings in solving a given problem once for 12 languages instead of solving it 12 times. Furthermore, there is a lot of power in a multilanguage environment because the programmer is free to choose the programming language based on which language is best for the application, and he is free to use existing program libraries regardless of what languages they are written in.

Future Directions

The VAX/VMS tools environment is still evolving. Some directions for future work include improving the integration between tools where suitable opportunities are perceived, providing fuller support for program design, providing better configuration management tools, and continuing the trend to increasingly distributed software development. The environment is likely to maintain increasing amounts of project data and to use that data for more kinds of project-control and reporting functions. The increasing use of workstations and their capabilities is another trend that will affect practically all tools in the VAX/VMS programming environment to one degree or another.

References

1. C. Mitchell, "Engineering VAX Ada for a Multi-Language Programming Environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, SIGPLAN Notices,* vol. 22, no. 1 (January 1987): 49–58.

2. D. Knuth, *The TeXbook* (Reading: Addison Wesley, 1986).

3. P. Gilbert, "Development of the VAX NOTES System," *Digital Technical Journal* (February 1988, this issue): 117–124.

4. G. Lupton, "Language-Sensitive Editor," *Digital Technical Journal* (February 1988, this issue): 28–39.

5. B. Beander, "VAX DEBUG: An Interactive, Symbolic, Multilingual Debugger," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging, SIGPLAN Notices,* vol. 18, no. 8 (August 1983): 173–179.

Anne Smith Duncan
Thomas J. Harris

Software Productivity Measurements

One objective of Digital Software Engineering is to build and maintain high-quality software products at reduced costs. To determine to what degree we are achieving this goal, Digital's Commercial Languages and Tools (CLT) Group is studying software productivity in relation to their software development cycle. In today's environment, engineers are building tools that assist in writing code and that automate project tasks. Further, development teams share processes and reuse existing code. To measure the effectiveness of these and other steps, this group has begun to devise software product and project metrics and to collect project data. To date, findings have been made for three metrics: engineering productivity, defect rate, and cost to build.

Digital's CLT Group builds and supports high-volume software products, including commercial language compilers, software development tools, and the VAX/VMS run-time library routines. CLT has been shipping native-mode VAX/VMS software products since 1978. Approximately one hundred software engineers, managers, release engineers, operational analysts, and system managers work in this group. User-documentation writers and editors, business product managers, and product marketing specialists are also members of the project teams.

The Importance of Productivity

Not long ago, computer users focused their attention on increasing the productivity of hardware because hardware was the component of greatest overall system cost. Although important, software development costs were small compared to the cost of running the software. Therefore, software engineers stressed writing programs that ran fast, used small amounts of memory and disk, and minimized the number of compiles and tests needed during the development process.

The Digital engineering culture allows each software project team substantial freedom to determine its own conventions, standards, and infrastructure. In this culture, moving a successful "process" from one completed project to a new one depended on the people who moved between projects. In the 1970s and early 1980s few supported tools were available, and tool development was done at the project level, if at all. Some processes were automated, most were not. Regression testing (testing that reveals whether something that previously worked still does) was done by hand, bug lists were compiled on blackboards, and debugging major integrations at base levels was difficult and time consuming. The project members paid minimal attention to tracing how and when things happened, and they documented this activity on paper, if at all.

Another aspect of this culture was the sense that each project team had to write all the code needed for that project. This attitude meant that code to do common routines was duplicated from project to project. Each team believed that its problem was unique, that it could not share code with any other team. The belief was pervasive that each problem was different and that each project team had found the only appropriate techniques.

By the mid-1980s, our customers, and our software engineers and managers started to pay much more attention to software costs, as the costs of software development and maintenance began to exceed the cost of hardware. Concurrently, certain trends both inside and outside Digital were forcing us to shift our focus from improving the hardware to improving the software development process. These trends were as follows:

- Marketplace expectations — Software customers were becoming more sophisticated and demanding. They needed software systems that would provide them with a competitive advantage in their marketplaces. They also wanted software that could be used safely by people of varied abilities and training.
- Increasing complexity — The complexity of developing software systems was increasing, and project management became more difficult as projects became interrelated and sometimes were located in different facilities, states, or countries. Communications between teams became increasingly difficult as the normal communications paths became clogged.
- New technology — New technologies were arriving at a faster rate and providing capabilities we had not considered feasible 5, 10, or 20 years earlier.
- Software maintenance — Various studies indicated that from 50 to 70 percent of the cost of software is spent on maintenance.[1,2] Maintenance includes defect correction, product support, and feature and capability evolution and extension. Software maintenance, especially defect correction and product support, consumes the human and hardware resources that should be used to build new products.
- Shortage of skilled software engineers — The growing demand for highly skilled and trained software developers, projected to continue for the next 20 years,[3] meant that experienced engineers had more pressure to increase their output.
- High-quality software — The demand for consistently high-quality software was increasing as more businesses built their operations around software systems. These businesses had little tolerance (nor should they have had) for software systems with defects.

To address these trends, Digital's software engineering managers and engineers have identified objectives for both the software product and the software development process.

- We want to build and deliver high-quality, dependable software products that meet our customers' needs in predictable, cost-effective ways.
- We want our engineers to solve new problems in creative ways, and we want to solve each problem only once.
- We want to reduce the costs of delivering new products.
- We want to reduce the costs of maintaining and supporting the product set.
- We want all team members and their managers to feel more "in control" of their own work.

And these objectives have to be accomplished within the constraints of our budgets and the availability of good software engineers.

In order to determine how to better accomplish these objectives, we needed to understand how we were doing at a point in time compared with how we had done in the past. This comparison is frequently described as measuring programmer "productivity" or measuring software engineering "productivity."

In February 1985, a graph was published under the topic of programmer productivity. The graph showed the actual and projected rates of growth in lines-of-code per programmer from 1980 through 1990.[4] It indicated that by 1990 the average software developer would produce 1,075 lines-of-code per month, up from 650 lines-of-code in 1985.

This graph re-emphasized to us the need to clarify how productivity should be defined and measured. Productivity in software development is more complex than simply increasing the lines-of-code produced by each programmer. The productivity of people, regardless of how it is measured, is only one part of the software development process. In any case, that projection caused us to seek answers to several important questions, such as the following:

- What exactly is programmer productivity or software engineer productivity?
- How can we help our software project teams to become more productive, and how can we measure whether or not their efforts and achievements are better?
- How do we know if our products and the ways in which we develop those products are better now than in the past? What do we mean by "better"?

The remainder of this paper describes some answers to these questions and how the answers were derived. A major benefit of this work has been the increased and continuing contribution by all members of the CLT group to more precise definitions of quality in our products and processes. Our findings indicate that CLT's productivity has improved over the last seven years. And our findings justify the costs of collecting and analyzing the data so that we can know whether we are continuing to do better work.

Software Productivity

Software productivity encompasses more than just the programming of software products. A software system is completed only when the functional and performance requirements have been met and when it is useful for the intended user. Therefore, the usual steps to reaching that state include analyzing the requirements; designing, coding, and testing the code; documenting the system for both its users and maintenance software engineers; and providing training and field support. The only way we can really examine productivity is to consider the software system in the context of the entire development cycle.

Two major dimensions of software engineering productivity are (1) the change in quantity of software produced for a given period of time at a given cost and (2) the quality of the resultant software system.

Since a software system is built to solve a set of problems, not as an end in itself, we need to consider the product and the process in the context of each other. Then we can measure productivity and use the results to help us focus on whether our process is better, and what needs to be changed in the development process.

The quality attributes that are important to the user of the software are important also to the engineer who supports and extends the software. For example, quality attributes include the usability, usefulness, defect level and rate, and performance of the software system. Also, we must include the quality attributes that affect future costs; for example, the ease of modifying to enhance or correct the software and the ease of porting the software to other hardware.

Influences on Productivity

A number of studies indicate that software productivity is influenced by multiple factors.[5] These factors include

- Personnel and team capabilities and experience
- Requirements on the resultant software system, including reliability, storage use, and performance
- Characteristics of the development process, including the use of disciplined engineering practices, the use of software tools, and the availability of hardware for development and testing

The major factors that have changed at Digital during the last seven years are (1) increases in size, complexity, and dependencies of products; (2) increased use of shared tools; and (3) the sharing (reuse) of design, code, and documentation between projects.

The first factor would be expected to decrease the overall productivity of the project teams. The second two factors would be expected to increase productivity.

Other papers in this issue of the *Digital Technical Journal* describe specific tools used during product development and give examples of design and code reuse.[6,7]

Tool Development and Use

In today's environment, each project team can choose whether to use tools, define its project infrastructure, and determine its own methods for running the project. With the availability of supported and useful tools, however, project members usually choose to automate some processes, thus avoiding the redundant effort of reinventing designs and code that already exist. The paper "VAX/VMS Software Development Environment" (this issue) describes the tools and their uses during the development process.[8] The same tools are used across multiple development phases. For example, the VAX DEC/CMS Code Management System tool, the version control system, can be used from the beginning through the end of the process. At the beginning, this program manages versions of the requirements documents; at the end, it manages the versions of code, tests, command files, and documents.

Here are some examples of CLT's use of these tools:

- Previously, the procedures for building and controlling source code were usually listed on a blackboard, in a notebook, or in someone's head. Now, the VAX DEC/CMS and VAX DEC/MMS Module Management System tools automate the versioning of source code, the identification of modules that belong to a particular base level or version, and the build processes. The library structures and MMS build procedures also serve as project documentation.

- Regression testing is now simplified by the VAX DEC/Test Manager software. By supporting attribute-based subset test selection, this tool makes it easier for software engineers working on optimizations or defect corrections to quickly run subsets of a major test system. Being easier, testing is done more often. Many projects routinely rebuild the project code (using CMS and MMS) and run either the entire test system or a part of it every night. The next morning, the software engineers know immediately if their previous work caused a new problem or regression. Projects that have adopted this process for builds and tests have almost completely eliminated the many hours of integration at base level.
- The VAX NOTES system, a distributed conference tool, helps in automating and tracking project design discussions and decisions. The project members can open a separate topic dealing with an issue. Subsequent responses from the project members, and perhaps field support personnel (world-wide), are available to all interested parties on Digital's internal network. Although the NOTES conference does not replace meetings of the project team for design discussions and reviews, it does provide an easy-to-use mechanism for describing the history of the discussions. NOTES helps to inform new project members of the project's history and status.

Reduced Redundancy

Our software engineers now search for code, designs, additional tools, and documentation that can be reused. Both managers and engineers consider reused code as an investment in design, programming, and testing that has already been paid for. Moreover, the support for that code has been planned and is in place. Reusable run-time components have been used and available since the first version of the VAX/VMS operating system in 1977. The VAX Common Run-Time Library (RTL) is used by all products. This library is a group of approximately one thousand software routines used by hundreds of software components and products for run-time support of common functions. Recently, major components outside the RTL have been planned and designed specifically to provide functions that are needed in multiple products.

Software Metrics

The best way to gauge improvements is to have a set of measurements that compares how things have changed over time. A software metric is a quantitative way to characterize an attribute of either the software system or the software development process. For a metric to be meaningful, there must be a way to measure these attributes consistently and objectively. Then various software systems and development projects can be compared to themselves over time and to each other. (That assumes other variables remain constant; for example, similar types of organizations building similar types of software using similar methods and processes.) Only when metrics have the same definition (and therefore are measured in the same way) should they be compared.[9,10,11]

Any metric process should guard against

- Measuring only one dimension; for example, quantity alone without considering quality; time only without regard for the product delivered (For the results of a study on the effects of measuring one dimension or criterion in favor of another, see Weinburg and Schulman's study on computer programming goals and performance.[12])
- Measuring for the wrong reasons; for example, using measurements to appraise an individual
- Comparing measurements with too many variables; for example, process control applications are different than payroll applications; the quantity of code per unit of time is less for high-level languages vis-a-vis assembly language code

In this paper, we discuss two sets of software metrics: product metrics to describe the software itself, and process metrics to reflect the process of software development.

Software Product Metrics

Product metrics (also called system metrics) describe the attributes of the software system or components of a system, and the related documentation, tests, and system control information (for example, command language batch streams). Size, usability, maintainability, number of defects, and performance are all attributes of a software system. For this study, we used three software product metrics: size, defects, and defect rate.

- The size S of a software product i is defined as

  $$S_i = \frac{S_e + S_c}{1000}$$

  in which S_e equals the number of lines of code, including data declarations, and S_c equals the number of lines of comments. Each line is counted as one regardless of the number of operators, operands, and comments that the line may include. Include files are counted once, and reused code shipped with the product is counted. Blank lines are not counted. Project tools, tests, test data, and control files are also not counted.

- The number of defects D for a product i is defined as

  $$D_i = D_b + D_d + D_r$$

  in which D_b is the number of customer reports answered as a "bug" or "correction given," D_d is the number of customer reports answered as "documentation error," and D_r is the number of customer reports answered "restriction on the use of the software."

- The defect rate DR_i, which also describes the software product, is defined as

  $$DR_i = \frac{D_i}{S_i}$$

  The defect rate provides a way to normalize the data associated with a particular product such that the defect rates of multiple products may be compared without regard for variances in product size.

Software Development Process Metrics

Process metrics describe the attributes of developing the software system, product, or component. Attributes of the process include the cost of development (in human resources, hardware resources, and calendar time), the predictability of the schedule and delivered software capability, the number of design and code reviews, and the length of time to respond to a customer inquiry or problem report. For this study we are using the cost and engineering productivity metrics.

- One definition for cost C of the development of the software product i is the length of time in months that the software engineers and project leader spent in the various phases of the project. Thus cost C is defined as

  $$C_i = DM_{iP1} + DM_{iP2} + DM_{iP3}$$

  in which DM equals the number of months directly charged to the project by the software engineers and project leader, and $P1$, $P2$, and $P3$ are phases 1, 2, and 3, respectively.

- The engineering productivity EP_i is defined as

  $$EP_i = \frac{S_i}{C_i}$$

  This metric provides a means for comparing various projects by normalizing the size and the cost of each project.

Indicators of Improvement

To answer the question, Are we doing better now than in the past?, we have to gather data on older projects and then compare it with data from more recent projects.

Collecting the Data

For the products in this study, we present three sets of data: size, number of defects, and development cost. We chose version 1 products and other major product versions in which more than 50 percent of the delivered code was new. All products in this study were developed on the VAX/VMS system, and all but one were written in the VAX BLISS-32 language. Shipment of these products to customers began during the period from late 1980 through summer 1987.

Collecting data from older projects was somewhat difficult: there were few common tools that we could use for data collection, files were frequently lost, and memories of the project team members were not always clear or accurate. Some projects did keep data, and these are included in the comparisons. Many projects of the late 1970s and early 1980s, however, did not collect or save needed data; so there are fewer data points for products shipped before 1983. Many of the early products developed in CLT were not written for the VAX/VMS system and have not been included either.

Collecting data from recent projects was easie because most of them used the same tools as part of the project infrastructure. We were able to collect product lines-of-code data in a consistent manner by using routines written in the VAX SCAN language to access the project

VAX DEC/CMS libraries. Customer-reported defect data has been collected for many years through a database that stores information from Software Performance Reports. The data for time and effort to build the products was collected from project phase review and accounting records.

The Results

This paper presents the findings for two derived metrics, the engineering productivity metric and the defect-rate metric, and the change in cost-to-build as reflected by the relationship between size and cost data.

The Engineering Productivity Metric

Engineering productivity indicates the rate of code production for an investment of each person-month. This metric is helpful for understanding whether programmer productivity is improving. Figure 1 shows this metric for 14 products and the date of the first shipment. The engineering productivity associated with the products delivered before January 1985 and those delivered since are shown by the regression lines. Since 1985 there has been a significant increase in the quantity of code produced for each developer-month. For the 1980 through 1984 period, the productivity rate ranged from 220 lines-of-code per developer-month to 1,487 lines-of-code per developer-month, with a mean of 792. For products shipped since January 1985, the productivity rate has ranged from 1,133 lines-of-code per developer-month to 3,735 lines-of-code per developer-month, with a mean of 2,169.

We attribute this improvement primarily to the increased reuse of code from other projects. Of the four most recently shipped version 1 products, reused components composed between 22 and 56 percent of the delivered code. Additionally, we believe that the use of common, supported tools that became available during the development of products delivered since January 1985 also contributed to this improvement.

The Defect Rate Metric

Defect rate is one measure of the quality of the software products that we ship to customers. Various published studies indicate that the "typical" defect rate for American industrial software is 10 defects per 1,000 lines of code,[13] and that this rate varies from 5 to 30 defects per KLOC.[14]

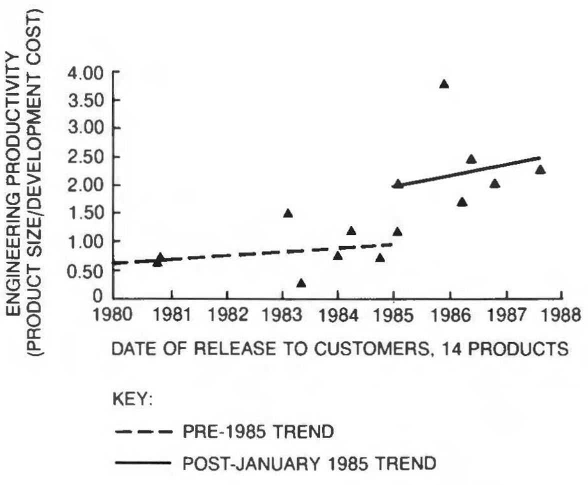


Figure 1 Engineering Productivity

To understand our own level of defects and to compare our performance to those published figures, we examined the post-release defect rate for 13 products over the time period from the date when it first shipped to customers until the summer of 1987. (One older product was eliminated because the accuracy of the data was suspect.) This data is shown in Figure 2. Since January 1985, the defect rate has decreased to 0.066 and less. Of the 7 products in the 1985–1987 grouping, 4 had zero customer-reported defects at the time of this study. Our pre-1985 defect rate ranged from 0.07 to 1.51 defects per KLOC.

Although this defect rate is advantageously low, the most important finding from this data concerns the trend of this rate: it is decreasing. That trend means that our software customers have more reliable software, and that we can reduce maintenance and support costs, and free engineers to work on new products as well as support other products.

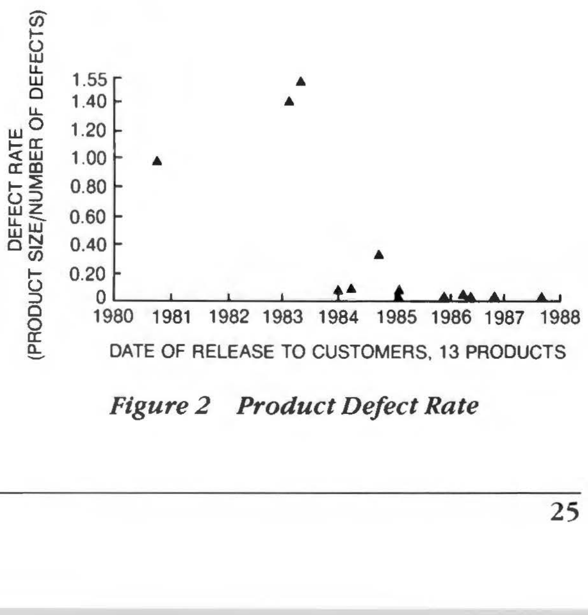


Figure 2 Product Defect Rate

We attribute the improvement in quality to an increase in the availability and use of tools, especially the VAX DEC/CMS, VAX DEC/MMS, VAX DEC/Test Manager, VAX Language-Sensitive Editor, and VAX Source Code Analyzer tools. (The last three tools first became available for internal Digital use during the second half of 1984.) During project quality reviews, each project team is questioned about the use of tools, which has led to an increased use of various tools during the development process. An unpublished survey taken of the current CLT project groups in the summer of 1987 indicated that 100 percent of the projects responding use the VAX DEC/CMS tool, 80 percent of the projects use or plan to use the VAX DEC/MMS tool, 100 percent of the projects use or plan to use the VAX DEC/Test Manager tool, 100 percent use interactive editors with 86 percent using the VAX Language-Sensitive Editor tool, and 79 percent use or plan to use the VAX Source Code Analyzer tool.

The Cost-to-Build Rate

We also examined the relationship between the size of the product and the cost to build it. Figure 3 shows the data for the cost to deliver tested and debugged code for 14 products. The horizontal axis indicates the project cost *C* for developing the product, the vertical axis represents the product size *S*, and the date at each data point is the year that the product began shipping to customers. Many studies show that the cost of software has a direct relationship to the size of the software produced.[5] Figure 3 indicates that CLT has delivered products with lower cost since January 1985 than for the period 1980 to 1984. For example, in 1980, one project cost $C = 145$ to design, implement, test and deliver a product with size $S = 83$. In 1987, a project delivered a product with a cost $C = 131$ with size $S = 294$. That is a 254 percent increase in product size with a reduction in cost of 10 percent. In other words, a product that is over three-and-one-half times the size of another was produced with less cost. We consider that as one indicator of improved productivity.

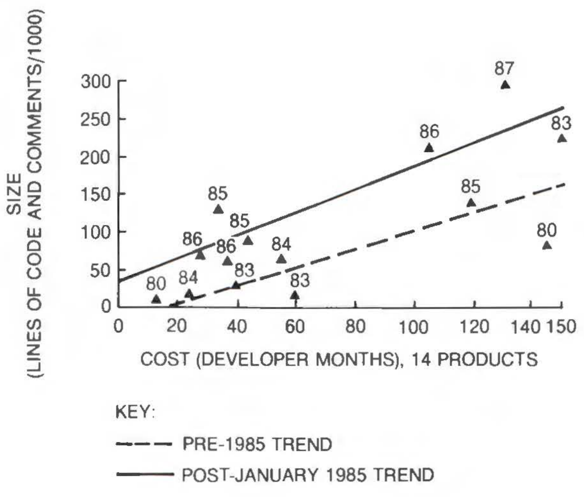


Figure 3 Relationship of Product Size and Cost

Continuing Improvement in the Future

By collecting project and process information, we have the data to compare past projects with new projects and to compare projects with themselves over time. We can use that data to evaluate the estimates and progress of current projects. This data helps both the project team and their managers to gauge how a project is doing. One use of the data in Figure 1 is to check the validity of the estimates of a new project. We can use this data to answer questions such as, Do the forecasted costs appear to be realistic given the history of past projects in this organization?

The data also provides a known base that can be used for comparisons with newer data when there are changes in the methods, tools, or training of engineers and their managers. Using these comparisons, we can determine if our process and products are getting "better" or not.

We are defining additional metrics, searching for those that add to our knowledge about the quality of the product and the process. Additional software product metrics include the mean-time-to-failure, module and product complexity, and maintainability and extendability of product code, documentation, and test systems. Additional development process metrics include the ratios of defects found by inspections, unit testing, pre-customer testing, and customer testing, the mean-time-to-fix a problem, the cost and effort for the various development phases, and the rate of successful test completion.

To collect the data, we use the software development tools that the project teams use as part of their project infrastructure. For example, the VAX DEC/CMS program maintains a complete history of the activity of a library, including additions and changes, when made, by whom, and for what reason. Using the VAX DEC/CMS history-file data, we can analyze the reasons for changes to modules and the rates of changes and deliveries for code, documents, and tests. As another example, we can track test failures and successes using the VAX DEC/Test Manager software. The test manager tool also provides a history of additions and changes to tests, thus yielding data about the arrival of tests into the test system.

Summary

The metrics and data discussed in this paper demonstrate that one group in Digital is building higher quality software products at lower costs. Particularly noteworthy is the increased quality of products, leading to reduced costs of maintenance. In several cases, one project team is able to support, maintain, and enhance one product, while providing support and maintenance for another.

One central question we asked earlier was, Do we understand what software development "productivity" means? Our answer is, More than we did in the past, but we have more work to do. We look at software development "productivity" to include more than the productivity of individuals or the project team. Software development productivity also includes the quality attributes of the product.

Quality and productivity improvement are ongoing. They have become part of our way of doing business. The managers and software development team members consider software metrics and measurements as additional tools that help them to manage projects. The ability to collect data from the productivity tools themselves has assisted in this process of change. Thus data collection has become a nonintrusive by-product of normal tool use. By using consistent collection methods, team members can compare the data across projects in the organization. However, that data is never used to measure an individual's productivity; the focus is always on the software development process and software systems that we deliver.

Acknowledgments

We want to express our appreciation to the present and past CLT project team members and managers for doing good work, for their unending search for improvement, and for their efforts with data collection.

References

1. R. Knight, "COBOL Still Strategic After All These Years," *Software News* 7(7) (June 1987): 58–64.

2. R. Hall, "Seven Ways to Cut Software Maintenance Costs," *Datamation* (July 15, 1987): 81–84.

3. "Help Wanted," *Business Week* (August 10, 1987): 50.

4. H. Davis, "Measuring the Programmer's Productivity," *Electronic Engineering Manager* (February 1985): 44–48.

5. B. Boehm, *Software Engineering Economics* (Englewood Cliffs: Prentice-Hall, 1981)

6. S. Greenwood, "VAX SCAN: Rule-Based Text Processing Software," *Digital Technical Journal* (February 1988, this issue): 40–50.

7. S. Grass, "Development of a Graphical Program Generator," *Digital Technical Journal* (February 1988, this issue): 101–109.

8. B. Beander, "VAX/VMS Software Development Environment," *Digital Technical Journal* (February 1988, this issue): 10–19.

9. T. Capers Jones, *Programming Productivity* (New York: McGraw-Hill, 1986).

10. S. Conte, H. Dunsmore, and V. Shen, *Software Engineering Metrics and Models* (Menlo Park: Benjamin Cummings, 1986).

11. R. Grady and D. Caswell, *Software Metrics: Establishing a Company-Wide Program* (Englewood Cliffs: Prentice-Hall, 1987).

12. G. Weinburg and E. Schulman, "Goals and Performance in Computer Programming," *Human Factors*, (16/1) (1974): 70–77.

13. B. Beizer, *Software Systems Testing and Quality Assurance* (New York: Van Nostrand, 1984).

14. W. Myers, "Can Software for the Strategic Defense Initiative Ever Be Error-Free?" *Computer* (November 1986): 61–67.

Glenn Lupton

Language-Sensitive Editor

The VAX Language-Sensitive Editor, a component of the VAX/VMS program development environment, is an advanced text editor specifically designed to help programmers develop and maintain program code. Developers of the product required that it include a simple interface that would be readily accepted by the VMS user community, language-sensitive features that improve programmer productivity, support for multiple languages with the same user interface, and support for user extensions. In addition, the editor had to mesh well with the existing program development environment offered by Digital. This paper provides the background of the development effort, a close look at the design of various features and some of the insights gained, and a summary of the current status and future directions for the environment provided by the editor.

Background

In late 1982, the Technical Languages and Environments Group started a project to specify a Programming Support Environment (PSE). Several Digital products related to supporting program development were already available or under development. The PSE project outlined a number of components needed to complete Digital's PSE offering. One component was an editor specialized for program development. The editors being used to develop software typically did not contain any special features to support the programming process. The PSE project developers saw this as an interesting opportunity, and the program editor became the first target of the PSE development effort.

Research of Program Editors

At the time, various universities were researching program editors, and a number of papers appeared in technical journals. The programming-related features the editors provided often varied with the language they supported, but they typically provided interactive syntax checking and special commands to insert language statements. Most of these were tree editors, which model a source file as a syntax tree, rather than text editors, which model a file as a stream of characters. An advantage of tree editors is that syntax errors and some semantic errors are precluded or diagnosed immediately. Tree editors can determine the syntactic context of the current editing position and offer assistance on the language constructs or the identifiers that are valid at that position. There are also operations that can be performed conveniently on a parse tree, such as cursor movement by syntactic element and elision, the suppression of selected program details in the display. Although tree editors can offer some very useful programming support, their disadvantages are significant. Their main drawbacks relate to their user interface, their performance, and their specialization to a single programming language or subset of a language.

Making modifications to source files is often awkward using a tree editor. Since tree editors insist that the contents of the file must always correspond to a well-formed syntax tree, there are serious constraints on the intermediate forms that the contents of the file can assume. In one tree editor, the language keywords are not items that can be edited by the user. A simple change, such as replacing the keyword WHILE with UNTIL, requires a number of steps, including saving the loop-body, deleting the WHILE-loop, and reconstructing it as an UNTIL-loop.[1]

Another tree editor copes with such difficulties by integrating a simple text editor with the editor.[2] Users may edit a portion of the source file as text by clipping a syntax subtree into the text editor. Of course, the benefits of the tree editor are not available when using the text editor facility. When returning the clipping to the source file, the tree editor parses the clipping to verify that its syntax is valid at the point where it is being inserted. In both of the above editors, there are constraints on how users make changes to source code. Moreover, they must think in terms of changes to a syntax tree, even if that is not the most natural way to think of a particular editing task.

Tree editors rely on pretty-printers to convert their internal parse tree representation of a source file into text for display to users. When the formatting style of the pretty-printer is agreeable to the user, this is a time-saver. However, every pretty-printing algorithm has limitations, including cases that the user can format more readably. Thus, another drawback of tree editors is that users must accept the formatting style of the pretty-printer even when they would prefer a different style.

Tree editors also consume considerable computer resources, both processing power for parsing and memory for storage of parse trees. Only single-user systems or lightly loaded, multiuser systems could accommodate the resource requirements of these editors.

Tree editors have difficulty in supporting certain language features, such as macros and conditional compilation. For example, the following fragment of C code uses conditional compilation to call the function PROCESS with an extra parameter when it is compiled for TARGET2:

```
          process(
                  input
#if target2
                  , length
#endif
                  );
```

Such language features pose considerable problems in both the construction of a parse tree and pretty-printing. Typically, tree editors place restrictions on the usage of such constructs.[3]

Another language feature that poses problems for tree editors is file inclusion. This feature inserts a specified source file into the compiler's input stream, temporarily suspending input from the original source file. The #include preprocessor control line in the C language is an example of this. Although a file specified by #include typically contains only declarations, the language does not place such a restriction on the contents of the file. Since the file may contain any fragment of a compilation, it might be impossible to construct a parse tree for the contents of the file, and so, impossible to edit the file using a tree editor. It may also be difficult to construct a parse tree for a source file that contains #include statements.

Still another drawback is that tree editors cannot be used for all editing needs. Most tree editors are tailored to a particular language. Users have to use other editors for other languages and for text files.

Product Requirements

The primary requirement of the program editor is to improve programmer productivity by supporting program editing with language-sensitive features. Based on the insights gained from the above research, the PSE project assembled a list of additional requirements and design considerations.

Ease of Use

Users must view the PSE editor as being easy to learn. In particular, since most of our customer base was using EDT, the program editor should have an interface that is compatible with EDT.[4]

Given that they know how to use EDT or another text editor, there should be very little that users have to learn in order to begin using the program editor productively.

Additionally, the PSE editor must not complicate the users' programming environment by forcing them to use the PSE editor for editing source files and another editor for other text files. They should be able to use the PSE editor as a replacement for their current text editor. The PSE editor must enhance the programming process with minimal changes to users' editing styles.

Multiple Language Support

The editor must support a variety of programming languages. FORTRAN was the most widely used programming language in our customer base; but many customers were using other languages, and a significant portion were using more than one high-level language. Some customers would soon start using the Ada language, and a program editor that would help users make the transition to Ada would be an important part of Digital's Ada Programming Support Environment (APSE). The editor must also make it easy for users to work on software written in more than one language and to work on multiple programs written in various languages. Therefore, the editor must support a variety of languages with a common user interface.

Flexibility
In addition to supporting a variety of languages, the editor must provide access to all the features of each language. The editor must not limit users to a subset of the language nor restrict the way in which language features are used. Users must be able to construct any legal program and format the code as they wish.

Extensibility
Users must be able to modify and enhance the editor to suit their preferences and their special needs. The popularity of EMACS, a programmable editor, was evidence of the need for this.[5] Also, many customers had tailored EDT for special uses.

In addition to providing a malleable editing interface, the editor would have to accommodate user-defined languages and user modifications to the languages provided by Digital. Users should not have to change their coding style or their project coding conventions when switching to the PSE editor.

Performance
Performance was an important design consideration. Some users complained that the performance of EDT was barely acceptable on a loaded timesharing system, which was the expected environment for the PSE editor. Developers working on a text editor for such a target system were then wrestling with the problem of echoing characters as fast as the user typed them. The PSE editor must support program editing without requiring significantly more system resources than typical text editors needed.

The EDITH Prototype

Although program editors that viewed a source file as a parse tree showed promise and had advantages over editors with a text view of a program, their disadvantages were significant. An alternate approach was to add language-sensitive enhancements to a text editor. This approach had been meeting with some success internally with enhancements to the EDT editor,[4] and externally with EMACS,[5] and the Z editor.[6] This is the approach that the PSE project chose to prototype.

Given the above requirements, the PSE project developed a prototype program editor called EDITH. This was a text editor with an EDT-style interface. It maintained only a textual representation of source code and could be used to edit any text file.

EDITH supported program editing by supplying templates for source constructs. A template is usually a skeleton for a language construct. Users can instruct the editor to insert a template into a source file. The following is an example of a template for an IF statement:

```
if {condition} then
    {statement}...
[elsif_part]...
[else_part]
end if;
```

Templates provide several benefits:

- Correct keywords and punctuation
- Proper formatting and indentation
- Consistent case conventions
- Source entry using fewer keystrokes

Templates typically contain syntactic markers indicating where other program elements can or must appear. Templates also aid the user in entering correct source code. Markers, such as [else_part] and {statement}, have templates or menus of templates associated with them that the user can select. The user is free to use the text editing capabilities of the editor to enter and modify program text. By providing language templates, the editor helps the user develop syntactically correct programs, without restricting the contents of the source file, as does a tree editor. This design for syntax support also performed well and could easily accommodate a variety of languages.

Additionally, EDITH interfaced to a parser to perform syntactic checking and to report errors. An explicit PARSE command would pass the source code to a language-specific parser. As errors were detected, the parser passed diagnostic information back to the editor. The editor displayed this information to the user, who could then step through the errors one at a time. As each error message was displayed, the editor positioned the editing cursor at the point in the source where the selected error was diagnosed. The user could then make appropriate corrections. This interface allowed users to find syntax errors without leaving the editor to run the compiler, and provided a convenient interface for reviewing the errors and locating the corresponding source code.

A number of ideas prototyped in EDITH evolved into features of LSE.

Foundation for the Implementation

Coding for the VAX Language-Sensitive Editor begin in January, 1984. At that time, a preliminary version of the VAX Text Processing Utility, VAXTPU, was available. VAXTPU is a text editor users can program using a procedural language. The language supplies a large number of built-in functions that are used to manage files, windows (portions of the terminal screen), text, and even subprocesses. VAXTPU provides compilation and execution facilities for programs written in the VAXTPU language. To use the VAXTPU editor, the user needs a user interface written in the VAXTPU programming language. The first release of VAXTPU provided two interfaces. One interface was an EDT emulator; the other, called EVE, was heavily oriented to the keyboard for VT200-series terminals. VAXTPU was to replace EDT as the VMS editor. Shortly after the release of VAXTPU, LSE would have to address compatibility with VAXTPU and EVE, as well as with EDT. To meet this new requirement, and the requirement for the editing capabilities to be user-tailorable, LSE would be a superset of VAXTPU. The high-performance design of VAXTPU would also help LSE meet its requirement to perform well on loaded timesharing systems. LSE would also use VAXTPU to provide EDT-compatible functions.

As a result, the LSE project used the VAXTPU sources as a base when implementing LSE. Although a number of the VAXTPU source modules had to be changed to interface to LSE functions, most of the VAXTPU code was used unchanged. Thus, LSE used the VAXTPU code as a library of run-time routines for terminal interaction and for support of the user-programmable features. This gave the LSE developers access to a powerful set of functions for text manipulation and screen management. At this point, the code for the EDITH prototype was abandoned.

Syntax Support

One obvious way an editor can support programming is to assist in the construction of syntactically correct programs. To provide such assistance, LSE adopted the template approach prototyped in EDITH.

Templates for Language Constructs

To provide support for entering source code, LSE adopted the template approach prototyped in EDITH. In a very basic sense, a template is simply text that is inserted into the editing buffer at the user's request. Within that text may be places where the user will have to enter additional text to complete the material.

In the context of a programming language, the text of a template is usually a sequence of lexemes, such as keywords and punctuation, that form some piece of the syntax of a programming language. Places where the user must fill in more syntax to have valid program text are indicated by syntactic markers called placeholders. Templates are inserted into a source file by an EXPAND operation. Text inserted as part of a template can be edited like any other text in the source file.

Tokens

The keyword or text that a user types to identify the template he wants inserted is called a token. An EXPAND operation replaces the token with the corresponding template. For example, typing IF into an Ada source file and then pressing the EXPAND key inserts the following into the file:

```
if {condition} then
   {statement}...
[elsif_part]...
[else_part]
end if;
```

A token is usually a keyword that introduces a language construct or the name of a predefined function call. Only one EXPAND key is needed to expand any token into a template. Users do not have to type the entire token name before pressing the EXPAND key, just enough to uniquely identify the token. If the prefix that they enter matches more than one token, the editor displays a menu of possibilities from which to choose.

This style of interaction has proved to be easy and effective. Users have access to a large number of templates, including templates for calls to all the VAX/VMS system services and run-time library routines, and can access any template quickly and easily.

Placeholders

As mentioned earlier, placeholders are the syntactic markers that are inserted into the editing buffer as part of a template. Placeholders are distinguished from program text by the brackets ([ ]) or braces ({ }) that delimit them. They are stored internally as text, and files that contain placeholders are simple text files that require no special processing for operations such as printing. Editor functions that operate on placeholders recognize them by the characters chosen as delimiters for the language. For languages in which braces or brackets are valid lexemes, other delimiters must be chosen. Typically, the delimiters in such cases are formed using braces and brackets in conjunction with some other character, such as a tilde (for example, {~statement~}).

Optional and Required Placeholders

A placeholder may represent optional or required language syntax. The optional or required nature of the placeholder is indicated by the enclosing delimiters. In the IF example above, the {condition} placeholder appears with braces, delimiters that indicate to the user and the editor that it is a placeholder for syntax that is required at that point. The [else_part] placeholder appears with brackets as delimiters, indicating that the else_part is optional. The placeholders terminating with ellipses (. . .) are called list placeholders. They indicate where users may enter more than one of the corresponding language constructs. Braces and brackets indicate whether a list placeholder is for required or optional syntax. Thus, {statement}. . . appears at a point where the user must enter one or more statements, whereas [statement]. . . appears where the user may optionally enter statements.

Expanding Placeholders

Like tokens, placeholders may be expanded. There are three types of placeholders: nonterminal, menu, and terminal.

Nonterminal placeholders expand into a template. For example, in the above IF template, the [else_part] placeholder expands into the following simple template:

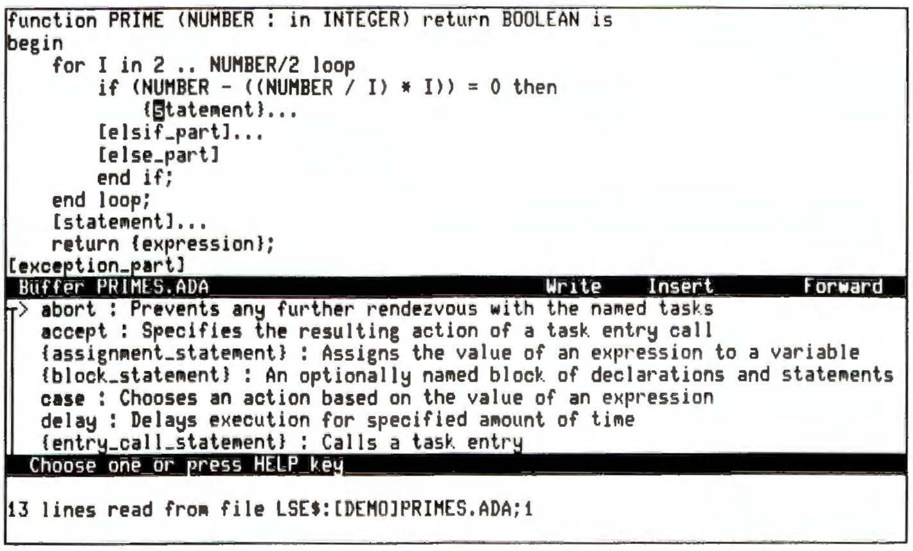
```
else
     {statement}. . .
```

A menu placeholder expands to a display of a set of choices. The {statement}. . . placeholder above is a menu placeholder. Expanding this placeholder results in the display of a menu of statements, as shown in Figure 1.

Expanding a terminal placeholder simply displays a description of the program syntax that must be entered at that placeholder. A typical example is the {identifier} placeholder; the expansion provides information such as the characters that are legal in an identifier and the maximum length of an identifier.

Filling in Placeholders

Once users have the level of detail they need from the templates, they can fill in the missing syntax



```
function PRIME (NUMBER : in INTEGER) return BOOLEAN is
begin
    for I in 2 .. NUMBER/2 loop
        if (NUMBER - ((NUMBER / I) * I)) = 0 then
            {statement}...
        [elsif_part]...
        [else_part]
        end if;
    end loop;
    [statement]...
    return {expression};
[exception_part]
 Buffer PRIMES.ADA                            Write    Insert         Forward
-> abort : Prevents any further rendezvous with the named tasks
   accept : Specifies the resulting action of a task entry call
   {assignment_statement} : Assigns the value of an expression to a variable
   {block_statement} : An optionally named block of declarations and statements
   case : Chooses an action based on the value of an expression
   delay : Delays execution for specified amount of time
   {entry_call_statement} : Calls a task entry
 Choose one or press HELP key

13 lines read from file LSE$:[DEMO]PRIMES.ADA;1
```

Figure 1 Editor Display with Choices for `{statement}. . .`

by typing text on the placeholders. In some cases, such as an assignment statement, users will likely be sufficiently familiar with the syntax of the language to type the complete statement on a statement placeholder, rather than to expand the statement placeholder and select the assignment template from the menu. In other cases there will be no more detail that can be supplied by the template, either because the template designer did not provide the detail or because the user has reached a true terminal in the language syntax such as {identifier}.

When text is typed on a placeholder, that text immediately replaces the placeholder. In the case of list placeholders such as {statement}. . . or {identifier}. . ., typing text on the placeholder causes a separator and a duplicate of the original placeholder to be inserted after the text. For example, typing COUNT on {identifier}. . . results in:

```
COUNT, [identifier]...
```

Note that the placeholder is now shown as optional and that the editor inserted a comma and a space as a separator. The definition of the identifier placeholder specifies the separator that should be used. The definition of a placeholder also specifies whether it should be duplicated vertically or horizontally. {identifier}. . . is an example of a placeholder that duplicates horizontally, whereas {statement}. . . duplicates vertically.

Erasing Placeholders

Many templates present placeholders for optional pieces of syntax that the user may not want. These optional placeholders can be erased using the ERASE PLACEHOLDER key. The algorithm for erasing a placeholder is one of the most complex algorithms in the editor. This would appear to be a simple text deletion, but even in the simplest cases some special rules come into play. In the following example, erasing the optional [expression] placeholder requires that a space be erased to prevent leaving an extra space before the semicolon.

```
return [expression];
```

Usually, the editor erases the whitespace (blanks and tabs) preceding a placeholder along with the placeholder. But when a placeholder appears at the beginning of a line, the editor must avoid erasing leading whitespace; erasing the leading whitespace would change the indentation of the line. In this case, it erases the whitespace that follows the placeholder.

In a number of cases, lines must also be erased. In the example of the IF statement shown earlier, after erasing [else_part] the editor must also erase the now blank line. This can be slightly more complicated, as in the following example of an IF statement in Pascal:

```
IF {expression}
THEN
    {statement}
[ELSE statement];
```

Here, after erasing the ELSE part of the IF statement, the editor must move the semicolon up to the end of the preceding line and erase the line that contained the ELSE statement. This example becomes more complicated when the preceding line is terminated by a comment. In that case, the semicolon must be inserted at the end of the statement placeholder but before the comment.

List placeholders, such as {identifier}. . ., require special handling as well. As mentioned above, typing the identifier COUNT on this placeholder results in:

```
COUNT, [identifier]...
```

When erasing the placeholder, the editor must erase the comma and space.

When erasing placeholders, the editor must maintain correct program syntax and proper formatting. The editor does not have a parser and pretty-printer for each language, but the algorithm for erasing placeholders achieves acceptable results by examining text surrounding the placeholder and using limited information about the language and placeholders.

On-line Help

The VAX/VMS operating system provides a HELP facility for managing and displaying information stored in a tree-structured text library. The information at a particular node in this tree is accessed by specifying a sequence of keywords. Each keyword selects a subtopic of the information for the preceding keyword. For example, issuing the VMS command HELP FORTRAN INTRINSIC COS accesses the information on the FORTRAN intrinsic cosine function.

Before LSE was developed, HELP libraries were available for a number of languages. LSE provides convenient access to this information by allowing the keyword sequence for accessing a particular node of the tree to be associated with a token or placeholder. For example, the token for the FORTRAN cosine function, COS, has the string "FORTRAN INTRINSIC COS" associated with it. When the user presses the language-help key while the cursor is on the token COS, the information for COS will display. Similarly, the {statement} placeholder can have a string associated with it that accesses a node of the tree whose subnodes describe the different types of statements for a language. This allows users to get help on language constructs without leaving the editor and without typing HELP commands.

Defining Language Support

The editor supports a special definition language for describing a language to the editor and specifying the tokens and placeholders. For each of the languages supported by Digital, the corresponding compiler project develops the language, token, and placeholder definitions, and the on-line HELP library. The definition language is documented for customers so they can modify the definitions supplied by Digital or add definitions for other languages. The editor accesses these definitions when it is invoked. By convention on the VAX/VMS operating system, source files for different languages are distinguished by a naming convention for the file-type portion of the file specification. The description of each language includes the file types that apply to that language. The editor uses this to determine which set of language definitions to associate with a source file. Users may edit several source files written in different languages in one editing session.

Templates Stray from BNF

A formal definition for the syntax of a language, such as a BNF description, provides a good reference when developing templates for a language. However, strict adherence to such a description can produce templates that are very tedious to use.

The syntax for a language may be defined using many intermediate productions. A straightforward conversion of this grammar into template definitions will result in a menu placeholder for each production that has several alternatives as a right-hand side. Therefore many menu placeholders will have elements that are also menus.

For example, to get from a placeholder for a statement to a template for a while-loop might require going from statement menu to control-statement menu to loop-statement menu to pretested-loop menu to while-loop. It would be much better to include the template for the while-loop in the menu for the statement placeholder. In general, templates can be vastly improved by eliminating intermediate menus and reducing the number of expansions required to access a template.

In some cases, the formal syntax definition for a language does not include some simple semantics that the templates should include. For example, the syntax may define BEGIN and END statements but not include their relation. The template for the BEGIN statement should include the matching END statement, plus placeholders for the syntax that may appear between them. There are a number of cases like this where common sense and coding standards should influence the template definitions. The template for a procedure declaration is another good example. Often a user site follows a coding standard that requires a procedure to have a specially formatted comment associated with it. Also, the coding standard might require that the body of the procedure be a BEGIN/END block, although the language does not require this. The predefined templates for a language can be a greater productivity aid if they are tailored to the way in which the language is typically used on a particular project.

Flexibility in Detail of Templates

The level of detail that templates provide is up to the author of the template definitions. For example, the placeholder {condition} could be a terminal placeholder that expands into a description of Boolean expressions in the language; or it could expand into more detailed templates and menus that provide the syntactic elements of a Boolean expression, such as AND, OR, and relational operators.

Highly detailed templates are especially useful in declarations, for example:

```
{identifier}... :
     [constant] [subtype_indication]
          := [initial_value];
```

Compared to a control construct such as an IF statement, the syntax of declarations is often complex, and likely to have a large number of options. Detailed templates help users with this complexity by presenting menus of choices and placeholders for various declaration options.

Observations on Using Templates

One of the significant aspects of the editor's support for entering source code is that its use does not interfere with the use of the editor for arbitrary text manipulation. There are no restrictions on the intermediate contents of the text buffer when reorganizing or restructuring code. Text manipulation operations that users have coded themselves in the VAXTPU language are also available. The final formatting of the source is up to users. Templates supply formatted language constructs, but users can reformat the program text without restrictions. This is not possible in most language editors.

In practice, users frequently take advantage of the source entry support afforded by templates. There are two primary reasons to use templates. One reason is for assistance with unfamiliar language syntax. Users who are just beginning to learn a new programming language can use the templates to help them get details correct. Users who are generally knowledgeable about the language can rely on the templates when they have to use a construct that they rarely have occasion to use, or when they have to make a call to a system routine that takes many parameters. A second reason to use templates is that they provide structured formatting and reduce typing. Even the most experienced programmers can benefit from using the editor to insert properly indented control constructs, matched BEGIN and END statements, and comment blocks, such as those used to describe the function, parameters, etc., of a procedure.

By helping users with language syntax and by inserting some of the source code for the user, the editor reduces errors and improves productivity.

Beyond Syntax — Integration for Programming Support

If assistance in entering syntactically correct source code were all that the editor offered with respect to support for program development, it would be a very useful productivity aid. However some of its most important productivity features do not deal with editing source code. In fact if the editor provided no support at all for language syntax, LSE would still be a valuable aid in program development because it also provides tight integration with the following facilities:

- VMS language compilers
- VAX Source Code Analyzer[7]
- VAX DEC/CMS (Code Management System)[8]

Compiler Interface

The editor provides users the means for diagnosing syntactic and semantic errors by interfacing with the language compilers. Each compiler for the languages that support LSE has been enhanced to generate diagnostic files that specify the compilation errors and the related source locations. The editor creates VMS subprocesses to perform the compilations. This is a very different design from the EDITH prototype that interfaced by procedure calls directly to a parser. This design has the following benefits.

- Users need not run the language processor from the editor.
- Concurrent compilations are possible.
- All compilation errors are diagnosed, not just syntax errors.
- No special parsers are needed.
- Implementation is straightforward.

Since the compilers place the diagnostic information in a file, the source processing does not need to be done from within the editor. Compilations can even be done overnight by a batch procedure, and the diagnostics reviewed later using the editor. Also, since the compilations are done in separate processes, the user is free to continue editing one source file while compiling others.

A drawback of using a parser, as in EDITH, is that the parser does not catch semantic errors, such as mismatched data types. These errors were caught by the compiler in a separate compilation step. In LSE, all compilation errors are caught by a single mechanism and can often be fixed in one step.

Another drawback of using a parser within the editor is the work required to develop the parser and keep it consistent with the compiler. Not all compilers are designed in a way that makes it easy to pick up the parser and use it as a separate, callable utility. Consequently, considerable time was saved by avoiding the implementation of a callable parser for each of the many languages LSE had to support.

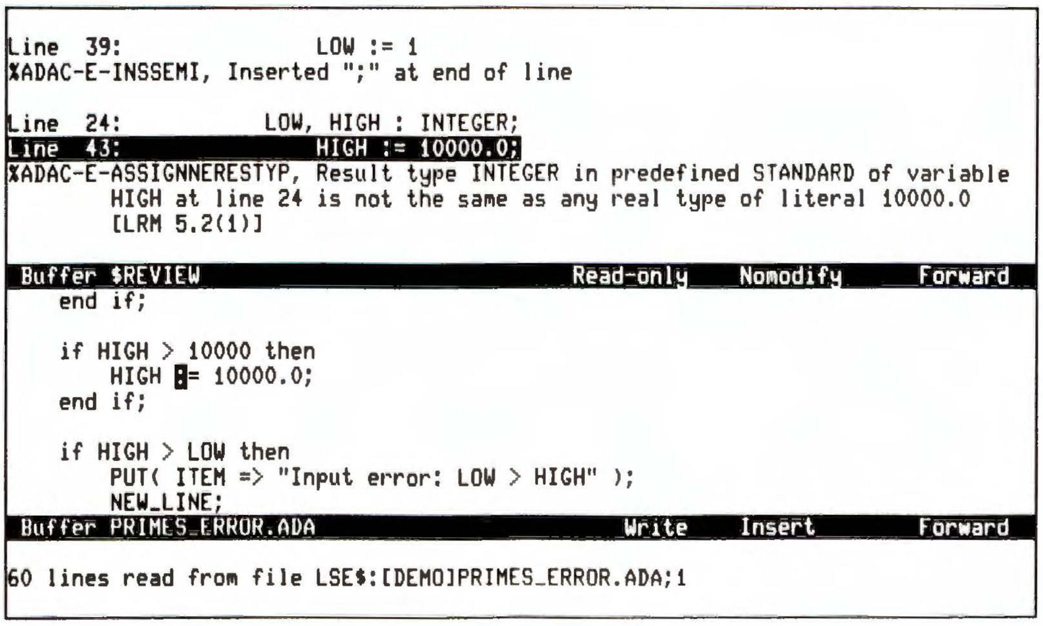
Once the compiler has written the diagnostics information to a file, the editor reads the file and displays the information in an editing window. Conventional editing commands for scrolling and searching can be used within this window. Special keystrokes allow the user to select a diagnostic and bring the corresponding source file into a second editing window, as shown in Figure 2.

Each diagnostic consists of message text and one or more lines of related source code. The message text may be one line or many lines. Usually one line of source related to the error is presented. The sophistication of the diagnostic information depends on the capabilities of the compiler being used. Most compilers present one-line error messages and the corresponding source line. The most sophisticated presents detailed error messages that include references to sections of the language manual and multiple related source lines. One use of multiple source lines is to display both the declaration and use of a variable with the error message. The user may select either source line in the diagnostic display, and the editor will position the editing cursor on the corresponding line in the appropriate source file. In some cases, such as a missing semicolon or a misspelled keyword, the diagnostic information supplied by the compiler will include a suggested correction. The editor will make the correction, subject to the user's confirmation.

Source Code Analyzer Interface

Perhaps the greatest productivity aid in LSE is its interface to the VAX Source Code Analyzer (SCA). SCA is a multilanguage, source code cross-reference and static-analysis tool. Compilers for supported VAX languages collect information during the compilation process and write this information to a special analysis file. SCA loads this information into a library for a program system. For each occurrence of an identifier in the program system, the SCA library contains information such as the source file, line, and column of the occurrence; the type of occurrence (read, write, declaration, call, etc.); and the class of identifier (procedure, variable, etc.). SCA can also display calls to and from a procedure, and can perform consistency checks on calls and declarations between compilation units.

Through its integration with SCA, LSE becomes a browsing utility for a software system. By positioning the editing cursor on a use of an identifier in a source file and pressing a key that invokes the GOTO DECLARATION command, the user can bring the source corresponding to the declaration



```
Line  39:                   LOW := 1
%ADAC-E-INSSEMI, Inserted ";" at end of line

Line  24:               LOW, HIGH : INTEGER;
Line  43:                   HIGH := 10000.0;
%ADAC-E-ASSIGNNERESTYP, Result type INTEGER in predefined STANDARD of variable
        HIGH at line 24 is not the same as any real type of literal 10000.0
        [LRM 5.2(1)]

 Buffer $REVIEW                                Read-only    Nomodify     Forward
    end if;

    if HIGH > 10000 then
        HIGH := 10000.0;
    end if;

    if HIGH > LOW then
        PUT( ITEM => "Input error: LOW > HIGH" );
        NEW_LINE;
 Buffer PRIMES_ERROR.ADA                           Write     Insert      Forward

60 lines read from file LSE$:[DEMO]PRIMES_ERROR.ADA;1
```

Figure 2 Editor Display with Source for Selected Diagnostic

of the identifier into an editing window. In this way, the user can view both the use and declaration on the terminal screen, as in Figure 3.

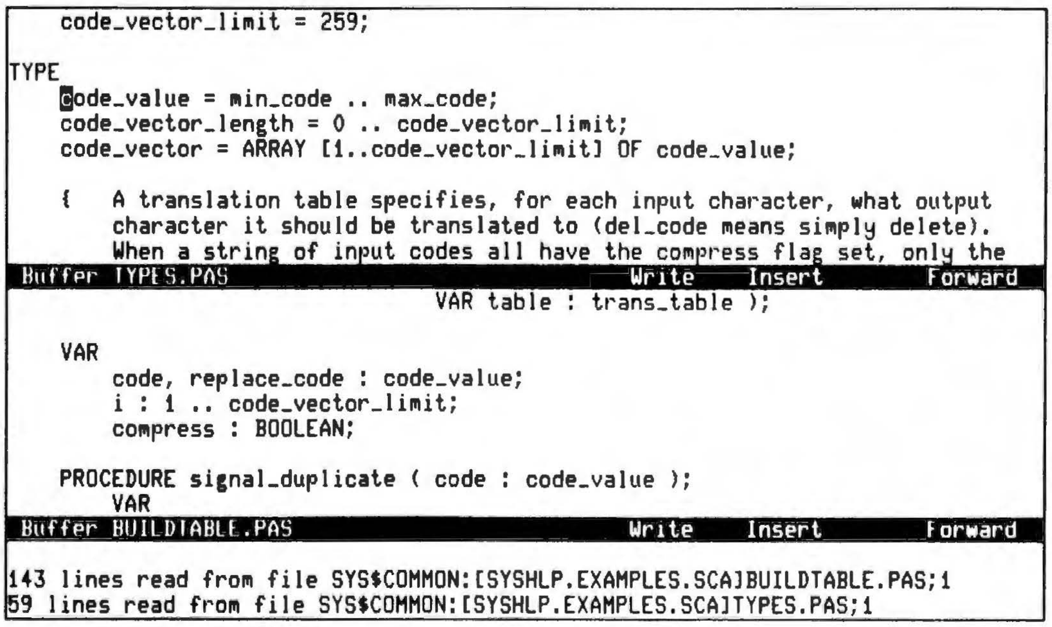
Users can perform more general queries using the SCA FIND command. Using FIND, a user can locate all the places where a variable is used. LSE displays the list of places and highlights the information for the first occurrence. Using single keystrokes, the user can select an occurrence and have the editor access and display the corresponding source, as shown in Figure 4.

LSE's integration with SCA has tremendously enhanced its value as a program development tool. Together these tools make programming tasks on a large software system much easier, since users do not need to rely on their memory or large cross-reference listings to locate declarations and uses of identifiers. The editor accesses the appropriate files and finds the source line of interest without the user even having to know the file name. By making it easier for developers to understand a software system, LSE with SCA speeds the development of new code and helps developers make changes to existing code more reliable.

VAX DEC/CMS — Code Management System

The VAX DEC/CMS tool stores project source files project, manages access to those files, and tracks changes to them. Source files are stored in a CMS library in a compressed format as changes to the original version of the file. Users reserve a file from the CMS library to make changes to it. After they are satisfied with their changes, they replace the file into the CMS library. Users may need to get a file from CMS without intending to change it. This is called fetching the file; a file that has been fetched but not reserved cannot not be replaced. Two users may reserve the same file at the same time. When the second replaces his version, CMS will assist in merging the two sets of changes. CMS provides many features for organizing a library, grouping sets of related files and related versions of files, maintaining variants of files, and inquiring into the status and history of files in the library.

CMS has become an important part of the daily routine for many VMS users. LSE provides complete access to all CMS commands from within the editor. LSE can also invoke CMS for the user. For example, when LSE needs to access a source file as part of the SCA GOTO DECLARATION function described above, LSE can invoke CMS to access the source file and bring it into the editor to display the specified declaration. Special LSE commands provide for reserving and replacing files while performing appropriate editor window and file management, eliminating intermediate steps for the user.



```
    code_vector_limit = 259;

TYPE
    code_value = min_code .. max_code;
    code_vector_length = 0 .. code_vector_limit;
    code_vector = ARRAY [1..code_vector_limit] OF code_value;

    {   A translation table specifies, for each input character, what output
        character it should be translated to (del_code means simply delete).
        When a string of input codes all have the compress flag set, only the
 Buffer TYPES.PAS                                Write    Insert         Forward
                                    VAR table : trans_table );

    VAR
        code, replace_code : code_value;
        i : 1 .. code_vector_limit;
        compress : BOOLEAN;

    PROCEDURE signal_duplicate ( code : code_value );
        VAR
 Buffer BUILDTABLE.PAS                           Write    Insert         Forward

143 lines read from file SYS$COMMON:[SYSHLP.EXAMPLES.SCA]BUILDTABLE.PAS;1
59 lines read from file SYS$COMMON:[SYSHLP.EXAMPLES.SCA]TYPES.PAS;1
```

Figure 3 Editor Display with Declaration and Use of `code_value`

The integration of LSE and CMS is an added convenience for LSE users that streamlines the usage of the two tools, and so enhances productivity.

Summary

The combination of templates, on-line help, and interfaces with compilers makes LSE an effective and easy-to-use program editor. Users can tune the templates provided with LSE for their own environment or define templates, help, and a compiler interface for languages that Digital does not support.

Through integration with SCA and CMS, LSE expands its capabilities from a source-code editor to a high-productivity user interface to the source code for a software system.

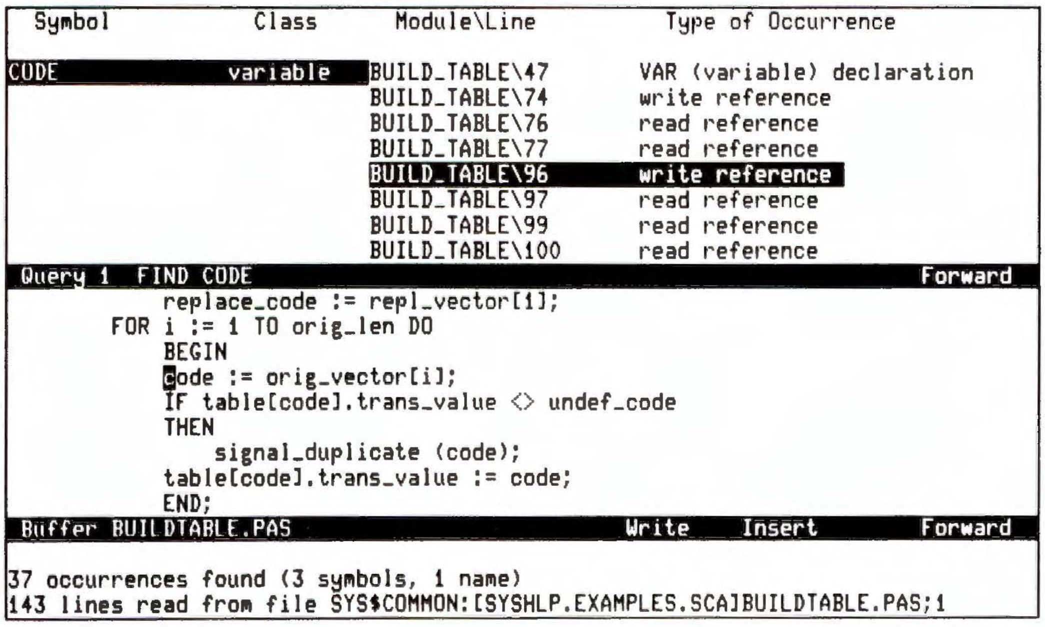
Since May 1985 when version 1 of the editor shipped, LSE has continued to evolve as a component of the VAXset, a set of productivity tools for software development. The VAX Language-Sensitive Editor is now playing an important role both as an editor designed to support writing software and as a hub for the integration of software development tools.

This paper describes the current release of VAX LSE, version 2.1.[7] LSE supports 15 languages including Ada, BASIC, C, COBOL, FORTRAN, Pascal, PL/I. In addition LSE supports some non-programming languages, including DATATRIEVE, a data management tool; DOCUMENT, a documentation markup language; and CDDL, a common data dictionary language.

Future Directions

The hardware environment for software development is shifting from timesharing systems to workstations. This shift will open some opportunities for LSE, making it possible for LSE designers to consider including capabilities that require more hardware resources than were typically available in the past. The current text-editor orientation does not preclude enhancements that require parsing the source, such as pretty-printing, cursor movement by syntactic element, and elision. The display capabilities of a workstation could support pretty-printing, using bold and italic fonts to improve the readability of source code. The availability of the mouse, icons, graphics, etc., allows for many enhancements to the user interface.

The software that supports program development will also continue to evolve. Additional lan-guages and tools can be expected to include LSE and SCA support, and there will be opportunities for LSE to grow through integration with other components of the programming support environment.



```
 Symbol             Class       Module\Line          Type of Occurrence

CODE             variable  BUILD_TABLE\47       VAR (variable) declaration
                           BUILD_TABLE\74       write reference
                           BUILD_TABLE\76       read reference
                           BUILD_TABLE\77       read reference
                           BUILD_TABLE\96       write reference
                           BUILD_TABLE\97       read reference
                           BUILD_TABLE\99       read reference
                           BUILD_TABLE\100      read reference
 Query 1   FIND CODE                                                   Forward
              replace_code := repl_vector[1];
          FOR i := 1 TO orig_len DO
              BEGIN
              code := orig_vector[i];
              IF table[code].trans_value <> undef_code
              THEN
                  signal_duplicate (code);
              table[code].trans_value := code;
              END;
 Buffer BUILDTABLE.PAS                       Write    Insert          Forward

37 occurrences found (3 symbols, 1 name)
143 lines read from file SYS$COMMON:[SYSHLP.EXAMPLES.SCA]BUILDTABLE.PAS;1
```

Figure 4   Editor Display with Selected "write reference" to `code`

References

1. T. Teitelbaum, T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM,* vol. 24, no. 9 (September 1981): 563–573.

2. M. Zelkowitz, "A Small Contribution to Editing with a Syntax Directed Editor," *Proceedings of the ACM SIGSOFT SIGPLAN Symposium for Practical Software Development Environments* (May 1984): 1–6.

3. J. Horgan, D. Moore, "Techniques for Improving Language-Based Editors," *Proceedings of the ACM SIGSOFT SIGPLAN Symposium for Practical Software Development Environments* (May 1984): 7–14.

4. *VAX EDT Reference Manual* (Maynard: Digital Equipment Corporation, Order No. AA-Z300A-TE, September 1984).

5. R. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," *Proceedings of the ACM SIGPLAN-SIGOA Symposium on Text Manipulation* (June 1981): 147–156.

6. S. Wood, "Z—The 95% Program Editor," *Proceedings of the ACM SIGPLAN-SIGOA Symposium on Text Manipulation* (June 1981): 1–7.

7. *Guide to VAX Language-Sensitive Editor and VAX Source Code Analyzer* (Maynard: Digital Equipment Corporation, Order No. AA-FY24B-TK, April 1987).

8. *VAX DEC/CMS Reference Manual* (Maynard: Digital Equipment Corporation, Order No. AA-L372B-TE, November 1984).

Stephen R. Greenwood

VAX SCAN: Rule-based Text Processing Software

The VAX SCAN product simplifies for programmers the building of software that recognizes complex text patterns. The product achieves this simplification by uniting powerful text-based pattern-matching capabilities with a procedural language that integrates these capabilities into the VAX/VMS environment. In typical applications of the product, programmer time to design, code, debug, and maintain programs is greatly reduced, contributing to increased software productivity. The short development time, low cost, and high reliability of the VAX SCAN product itself is attributable to the procedures and tools available in Digital's engineering environment.

The text processing capabilities typically provided by high-level languages are quite primitive, especially in contrast to the syntax diagrams used in describing programming languages. High-level languages normally include operations to move, compare, concatenate, search, and perhaps sort text strings of fixed length. Syntax diagrams, more formally known in computing literature as grammars, present a very nice model for very complex text patterns.

Programmers throughout the computing industry understand both primitive text operations and the more complex patterns represented by syntax diagrams. However, few programmers use syntax diagrams to express input to a program, because of the complexity involved in building the software needed to recognize such patterns.

The VAX SCAN product solves the complexity of using syntax diagrams by making text pattern recognizers available to the general VAX programming community. The SCAN language allows a user to express language grammars in a form very similar to the BNF-like syntax diagrams that appear in VAX/VMS documentation. The product builds a recognizer, in the form of an object module, that matches text described by the grammar. In addition, the product supports the VAX Common Language Environment, so that VAX SCAN recognizers can be easily integrated into existing programs.

Principles of the Language

The text processing capabilities of the SCAN language are based on the simple substitution paradigm of finding a specified pattern and replacing it with a string of text, much like a substitution command in an editor. SCAN differs from most editors, however, in that the search pattern can be as complex as a grammar for a programming language. The replacement text can be generated by an arbitrarily complex sequence of procedural statements. The actual SCAN language construct that supports this substitution paradigm is called a macro.

The syntax of a macro is given as follows:

```
MACRO macro_name attribute...
        { pattern } ;
    macro_body
END MACRO;
```

In a macro, the `pattern` specifies the text that the macro is to search for and match. The text matched by the pattern is then replaced by text generated by the `macro_body`. Figure 1 is an example of a macro that searches for time values and replaces them with a string.

In this example the pattern describes a time value as an integer, followed by a colon, followed by a second integer, followed optionally by another colon and a third integer. Whenever this pattern is matched, the VAX SCAN product replaces that occurrence of the pattern with the text generated by the macro body. The algorithm for generating the replacement text increments a static variable named `count` and specifies the replacement text with a special SCAN statement called the ANSWER statement. The first time the macro is invoked, the replacement text will be the string "`time: 1`". The second invocation of the macro produces the replacement text "`time: 2`". The TRIGGER keyword determines the type of macro, described later in the section Invoking Macros.

The body of the macro in this example is written using the procedural portion of the SCAN language. This portion not only serves as a means of creating replacement text, but also allows the language to achieve integration with the VAX/VMS environment. In that respect, SCAN is similar to PASCAL.

The procedural portion of the SCAN language includes procedures, functions, statements, and data structures that a programmer can use to interface with procedures written in other VAX/VMS languages. Procedures that can either call or be called by routines written in other languages can encapsulate SCAN's text processing capabilities. This encapsulation permits the VAX/VMS user to conveniently access the unique capabilities of the SCAN language.

Macros are the central construct of the SCAN language. Some additional considerations must be discussed, however, to fully understand the language:

- The origin of the language
- The invocation of macros
- The specification of patterns
- The method by which text matched by the pattern can be viewed by the macro body
- The interplay between the procedural code and text scanning

Origin of the Language

Few languages are designed from scratch; SCAN is no exception. The text processing paradigm employed by the SCAN language is an outgrowth of techniques developed for parsing computer languages using context-free grammars. These techniques permit the syntax of a language to be described by a grammar. Efficient algorithms are then used to check that an input stream of characters conforms to the syntax specified by the grammar.[1]

The MACRO language developed at Sperry Univac Corporation for their 1100 Series computers took this concept and applied it to string substitution and, in fact, developed the notion of macros that are found in the SCAN language.[2] The PASCAL language and the VAX/VMS architecture also influenced the design of the SCAN language. SCAN's data structures and data types are based on concepts in PASCAL. The desire for SCAN to integrate well with the rest of the VAX/VMS environment mandated that it be convenient to call SCAN programs from other VAX/VMS languages.

Invoking Macros

Several principles influenced the rules used by the SCAN language to invoke macros. Since a pattern might appear numerous times in a file of text, the first principle was to apply a macro each time its pattern is found so that the macro can make multiple transformations, just like an editor substitution command. Second, it was important

```
MACRO find_time TRIGGER { integer ':' integer [ ':' integer ] };

    DECLARE count: STATIC INTEGER;

    count = count + 1;
    ANSWER 'time: ', STRING( count );

END MACRO;
```

Figure 1 Example of a Macro

to be able to specify many concurrent substitutions. Therefore, when a VAX SCAN program is compiled, linked, and then run, it behaves much like a batch (rather than an interactive) substitution command. The third principle in designing macro invocation rules was that patterns may be of varying degrees of complexity. To manage very complex patterns, the patterns have to be separable into parts, much like a program can be separated into multiple procedures.

Two types of macros are defined to support these principles: syntax macros, and trigger macros. These names reflect the two different macros distinguished by the attributes SYNTAX and TRIGGER within a SCAN macro declaration.

Trigger macros provide search and replace semantics. A program may contain any number of trigger macros. Upon executing a START SCAN statement, a program begins scanning the input stream of text specified by that statement for the patterns specified by the trigger macros. Figure 2 illustrates a series of trigger macros.

The file `file_to_be_scanned.dat`, in this example (specified in the START SCAN statement) is searched for `a_pattern`, `b_pattern`, and `c_pattern` (specified in trigger macro patterns). The file `file_to_be_created.dat` (also specified in the START SCAN statement) is created by the program. Its contents will be the original file, `file_to_be_scanned.dat`, including the substitutions performed by the macros.

Syntax macros permit patterns to be defined in terms of other patterns. That is, within one pattern a programmer can request that a pattern described by a separate syntax macro be recognized. This concept of composing a pattern from more elementary patterns is the basis of formal language theory.[1] This topic is explored in greater depth in the next section.

Specifying Patterns

The design of patterns is influenced by the following three sources:

1. The BNF-style syntax diagrams customers find in Digital's manuals

```
.
.
.
MACRO find_pattern_a TRIGGER { a_pattern };
    ANSWER replacement_text;
END MACRO;

MACRO find_pattern_b TRIGGER { b_pattern };
    ANSWER replacement_text;
END MACRO;

MACRO find_pattern_c TRIGGER { c_pattern };
    ANSWER replacement_text;
END MACRO;

PROCEDURE main_procedure MAIN;
    START SCAN
        INPUT FILE 'file_to_be_scanned.dat'
        OUTPUT FILE 'file_to_be_created.dat';
END PROCEDURE;
.
.
.
```

Figure 2 Trigger Macro Example

2. Compiler theory for context-free grammars
3. The original MACRO language implemented by Sperry Univac Corporation

All three sources recognize a two-level approach for specifying patterns. The lower level recognizes simple constructs, such as numbers, keywords, names, and punctuation. The higher level arranges the lower level constructs into statements and groups of statements.

This two-level approach has several advantages. Abstracting primitives into lower level patterns results in a more uniform use of primitives. That, in turn, makes the overall pattern easier to remember, which is of great benefit to a programmer trying to design a language that he is trying to recognize using the VAX SCAN product. Compiler theory also states that lower level patterns can be recognized very efficiently if they conform to a set of rules, such as regular expressions.[1]

To take advantage of those features, the SCAN language provides a two-level description of a pattern. The lower level pattern, called a token, groups characters. The higher level pattern that appears in a macro is composed of tokens.

Figure 3 illustrates several SCAN tokens. The syntax for the patterns comes largely from the conventions used in Digital's software manuals.[3] Curly braces surround a sequence that is required, and square brackets surround an optional sequence. An ellipsis indicates that the prior sequence can occur multiple times, and a vertical bar separates alternative choices. Thus in the Figure 3 example, `morse_code_letter` describes a required sequence of " . " or "_" characters. This required sequence can occur one or more times.

The token `identifier` illustrates the pattern for an identifier in the SCAN language. The token definition uses two sets, `alpha` and `other`, to simplify the specification of its pattern. The definitions of these sets appear in the example as well.

In macros, higher level patterns are defined using the same operators that are used in token declarations. Unlike a lower level pattern, however, the operands of a macro pattern are tokens and other macros, rather than characters and sets. Therefore, tokens are the building blocks of a macro pattern. Referencing a macro within a macro pattern provides a subroutine-like capability for patterns. The placement of a macro name in a macro pattern indicates that the pattern of that macro should be recognized at the point of reference.

Defining the pattern of one macro in terms of other macros adds significant power to SCAN's patterns. This power, illustrated in Figure 4, shows the syntax for a SCAN token declaration. The pattern found in a token declaration is equivalent to patterns that can be described using regular expressions. Token patterns need to express the precedence of three operators: alternation, concatenation, and repetition. In addition, a token pattern supports nested subpatterns that are either required or optional.

This example utilizes macros to provide

- Levels of abstraction
- Sharing of patterns
- Recursion

Levels of abstraction refers to the process of building a hierarchy of concepts in which each

```
TOKEN morse_code_letter { { '.' | '_' }... };

TOKEN begin_keyword { 'BEGIN' };

SET alpha ( 'a'..'z' OR 'A'..'Z' );
SET other ( '0'..'9' OR '$' OR '_' );
TOKEN identifier { alpha [ alpha | other ]... };
```

Figure 3   Token Examples

level of the hierarchy is built on the next lower level. The SCAN language justifies two levels of patterns, tokens and macros, based on this principle. In Figure 4, the principle is employed further. `Token_declaration` is defined in terms of `token_pattern`, which is defined in terms of `token_operand`. Each macro describes a level of abstraction that makes a complex pattern both simpler to write and to understand. In this particular example, the levels expose the precedence of the three token pattern operators. `T_pat3` and `token_pattern` are examples of sharing patterns by means of macros. Much as a subroutine is a vehicle for sharing code in a FORTRAN program, macros are a vehicle for sharing a pattern in SCAN.

Recursion — defining a pattern in terms of itself — is very useful in the description of many patterns, especially when patterns can be nested as in the case of a token pattern. In Figure 4, `token_pattern` is defined in terms of `token_pattern` because token patterns can be arbitrarily nested using curly braces and square brackets.

```
TOKEN token_keyword....     ! patterns for the tokens
TOKEN identifier....        ! have been omitted
TOKEN character_string....
TOKEN ';'....
TOKEN '¦'....
TOKEN '['....
TOKEN ']'....
TOKEN '{'....
TOKEN '}'....
TOKEN ';'....
TOKEN '...'....

! syntax for token declaration

MACRO token_declaration SYNTAX
        { token_keyword identifier '{' token_pattern '}' ';' };

! syntax for alternation

MACRO token_pattern SYNTAX { t_pat3 [ '¦' t_pat3 ]... };

! syntax for concatenation

MACRO t_pat3 SYNTAX {t_pat2...};

! syntax for repetition

MACRO t_pat2 SYNTAX { t_pat1 [ '...' ]};

! syntax for optional and required patterns

MACRO t_pat1 SYNTAX { token_operand
                    ¦ '[' token_pattern ']'
                    ¦ '{' token_pattern '}' };

! syntax for operands

MACRO token_operand SYNTAX { character_string ¦ identifier };
```

Figure 4 Syntax Macro Example

Viewing Matched Text in the Macro Body

Macro patterns provided a simple mechanism for describing complex patterns. Macro bodies provided a powerful mechanism for creating replacement text. For a complete solution, the SCAN language needed a robust means so that a macro body could view the text matched by the pattern.

A solution to this problem was provided in the original MACRO language. That solution consisted of inserting variables in the pattern to capture the text matched by a segment of the pattern. See Figure 5 for an example.

`Hour`, `minute`, and `second` are variables inserted to capture matched text. `Hour` is assigned the text matched by the first integer token. `Minute` is assigned the text matched by the second integer token. `Second` holds the text matched by the third integer token, which may be the null string if the optional pattern `[ ':' integer ]` is not matched.

A pattern can, in fact, contain an arbitrary number of variables, each of which can capture the text, line number, and column position of one or more tokens matched by the pattern.

Interplay between Procedural Code and Text Scanning

The SCAN language is an interesting amalgamation of a rule-based language and a procedural language. The text scanning capabilities of the SCAN language are rule-based. Tokens and macros describe the rules for recognizing patterns. These descriptions have no concept of flow of control. Macro bodies, on the other hand, are algorithms with a very definite concept of flow of control. The interaction of the rule-based and procedural parts of SCAN is a key principle of the language.

A SCAN program starts executing in procedural mode at the start of its main procedure. That main procedure may be a SCAN main procedure or one written in another VAX/VMS language. The rule-based technique of pattern matching begins with the execution of a START SCAN statement. The START SCAN statement specifies the input stream to be scanned by the macros and the output stream to hold the transformed input stream. The main procedure in Figure 6 is a SCAN main procedure named `main_proc`. The START SCAN statement commences the search for the pattern described by the macro `translate_morse`.

```
CONSTANT h = 'hhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhhh';
CONSTANT m = 'mmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmmm';
CONSTANT s = 'sssssssssssssssssssssssssssssssssssssssssssssssss';

! replace a time such as 5:45:01 with h:mm:ss
! where number of h's, m's and s's corresponds to the number
! of digits in the time

MACRO find_time TRIGGER
    { hour: integer ':' minute: integer [ ':' second: integer ] };

    ANSWER h[ 1..LENGTH( hour )], ':';
    ANSWER m[ 1..LENGTH( minute )];
    IF second <> ''
    THEN
        ANSWER ':', h[ 1..LENGTH( second )];
    END IF;

END MACRO;
```

Figure 5   Variables for Capturing Matched Text

Each time a macro matches its specified pattern, procedure mode is entered for the duration of the execution of the macro body. When the macro body completes execution, the text replacing the matched text will have been generated and can be substituted in the output stream for the matched text.

Eventually, the input stream will be exhausted and the output stream completes. At this point, execution continues with the statement following the START SCAN that initiated pattern matching, and the program returns to procedural mode.

Software Productivity Benefits of Using VAX SCAN

Increasing software productivity depends on reducing the cost of implementing software while maintaining a high degree of reliability.

Within its problem domain, the VAX SCAN product increases programmer productivity by dramatically simplifying the solution to a problem. Consider the following program fragment:

```
TOKEN space { { ' ' | s'ht' }... };

MACRO compress TRIGGER { space };
    ANSWER ' ';
END MACRO;

PROCEDURE main_proc MAIN;
    START SCAN
        INPUT FILE 'input_logical'
        OUTPUT FILE 'output_logical';
END PROCEDURE;
```

This short sequence expresses the concepts of opening a file, scanning it for arbitrary-length

```
MODULE morse_code;
    DECLARE letter_count, error_count : INTEGER;
    EXTERNAL PROCEDURE
        morse_to_ascii ( DYNAMIC STRING, FIXED STRING(1) ) OF BOOLEAN;

    TOKEN morse_letter { { '.' | '_' }... };

    MACRO translate_morse TRIGGER { dots: morse_letter };
        DECLARE char : FIXED STRING(1);
        IF morse_to_ascii( dots, char )
        THEN
            ANSWER char;
        ELSE
            error_count = error_count + 1;
        END IF;
        letter_count = letter_count + 1;
    END MACRO /* translate_morse */;

    PROCEDURE main_proc MAIN;
        letter_count = 0; error_count = 0;
        START SCAN
            INPUT STRING '...._ ._ _..._ / ... _._. ._ _.'
            OUTPUT FILE 'sys$output';
        WRITE letter_count, error_count;
    END PROCEDURE /* main_proc */;

END MODULE /* morse_code */;
```

Figure 6 A Complete SCAN Program

sequences of contiguous blanks and tabs (s'ht' is the SCAN notation for a horizontal tab), replacing the matched sequence with a single blank, and placing a copy of the modified input file in an output file.

The sequence is at least one order of magnitude shorter than an equivalent algorithm in PASCAL. In general, using SCAN decreases the times to design, code, debug, and maintain programs.

The domain of programs well suited for implementation using the VAX SCAN product is somewhat difficult to assess. In this section, several types of applications are surveyed to give an idea of the range of applications and the cost of implementing each.

A typical use of the VAX SCAN product is to create a tool that will extract information from a set of files. An example of an extractor is a program to read a VAX SCAN source file and report the numbers of blank lines, the lines containing just comments, and the lines containing code. A version of such an extractor was written to gather statistics for this paper. The program prompts for a file specification, which may contain wild-card characters, and then scans all the files that match the file specification. The program lists the statistics for each file matched and the totals for all files scanned. The extractor consists of 50 lines of code, 25 blank lines, and 3 lines of comments; the entire extractor program took approximately 30 minutes to create and debug.

Another typical use is to build translators. Translators make modifications to parts of a file and leave the rest of the file unchanged. During the development of the SCAN language, several changes were made to its syntax. For example, parentheses were changed to braces in token and macro patterns, and files became explicitly declared rather than implicitly declared objects. Rather than change all the programs in the test system manually, we wrote a translator to do the job. The program consists of 188 lines of code, 60 blank lines, and 7 comment lines. Like the previous example, the program converts all files that match a specified file specification and reports which files were modified.

A third example that merges the concepts of an extractor and a translator is a program that reads BASIC source files that may contain record declarations. For each BASIC record declaration encountered, the SCAN program outputs an equivalent VAX Common Data Dictionary declaration. This program consists of 207 lines of code and 71 lines of comments and took approximately one work day to write and debug. Sample input and output for this record translator is given in Figure 7.

A language with a limited problem domain has the potential to decrease programmer productivity, because such a language is unlikely to be a programmer's main implementation language. Thus a programmer has a tendency to forget the details of the language, which in turn decreases his efficiency.

```
      Input                         Output

200 RECORD example             example STRUCTURE.
      BYTE a,b                     a DATATYPE SIGNED BYTE.
       INTEGER WORD c(5)           b DATATYPE SIGNED BYTE.
      GROUP nested_group           c DATATYPE SIGNED WORD ARRAY 0:5.
         HFLOAT d(10,10)           nested_group STRUCTURE.
         STRING e=5                    d DATATYPE H_FLOATING ARRAY 0:10 0:10.
         REAL f                        e DATATYPE TEXT SIZE 5.
      END GROUP                        f DATATYPE F_FLOATING.
    END RECORD                     END nested_group STRUCTURE.
                               END example STRUCTURE.
300 END
```

Figure 7   Sample Input and Output for Record Translator

VAX SCAN attempts to minimize this problem by

- Basing pattern matching constructs on syntax diagrams used in Digital's documentation
- Providing a VAX Language-Sensitive Editor interface for creating and compiling SCAN programs
- Adding support for the VAX/VMS Debugger, including features to monitor pattern matching
- Providing simple integration with procedures written in other VAX/VMS languages
- Basing the syntax on a well-known language, PASCAL
- Providing an extensive on-line help facility

In summary, the benefits of using the VAX SCAN product are twofold. First, for a large class of problems, SCAN drastically reduces the cost of designing, implementing, and maintaining the solution to those problems. Second, the product contains many ease-of-use features to minimize the overall cost of software development.

Leveraging the VAX SCAN Implementation

The initial version of the VAX SCAN product took approximately three developer-years to produce. This period includes the time to design the language, design and implement the VAX SCAN compiler and the run-time library, and internally test the product. The initial version was ready to be field tested by customers at the end of the three-developer-year effort.

Producing an optimizing compiler and run-time library in only three developer-years is a significant accomplishment. Three factors in Digital's engineering environment contributed greatly to making this level of software productivity possible.

- Digital's Software Development Policies and Procedures
- The VAX/VMS software environment
- The VAX/VMS tool set

Digital's Software Development Policies and Procedures

Digital's Software Development Policies and Procedures define the life cycle of a standard Digital software product.[4,5] This life cycle is divided into five phases.

- Phase 0 — Requirements gathering
- Phase 1 — Design of the product
- Phase 2 — Implementation of the product
- Phase 3 — Field testing of the product
- Phase 4 — Maintenance of the product

Each phase has a set of inputs and outputs. The output of Phase 0 is a product-requirements document that becomes an input to Phase 1. The output of Phase 1 includes a description of the product at a level that can be documented, a design of the product to a level at which the implementation costs can be estimated, and a development plan describing Phase 2 as a set of scheduled tasks.

These procedures and policies greatly reduce the effort required to manage a project by providing a common model for describing a project. The procedures provide a common framework and terminology to address issues of scheduling, cost, and task completion. A project manager, his team members, and people from other supporting groups, such as field test administration, all measure their efforts in terms of these common parameters. Since the definitions of tasks are specified as standards, time is not wasted discussing differences of opinions about these definitions. As a result, the different groups that develop, field test, and manufacture the product can work rather independently. The phase process details at what times those groups need to coordinate their activities and the manner in which they will do so.

Many of the documents that are part of the phase process have markup language templates. Each template outlines the contents of the document and includes a series of questions and checklists of the items to be considered during the preparation of the document. The questions and checklists are a distillation of the successes and failures of previous software products; the use of these questions and checklists helps to reduce the number of unanticipated problems that often plague software development.

Digital's Software Development Policies and Procedures provided a platform on which the VAX SCAN team built their product. By following these guidelines, less time was spent managing the project, leaving more time for the design and development of the final product.

VAX/VMS Software Environment

At Digital, software engineers have always strived to achieve a high degree of integration among VAX/VMS products. For example, command line parsing for most products is provided by a common command line utility. The use of the VAX RMS product as the standard I/O package is another example. VAX languages foster integration by supporting parts of an application being written in different languages. To meet this level of integration, Digital has evolved standards that describe

- Calling sequence
- Format of data types
- Format of descriptors
- Alignment of record fields
- Processing of exceptions
- Interface to the debugger
- Management of dynamic storage
- Structure of files

For a product like VAX SCAN, each standard represents a design problem that had already been solved. The VAX SCAN product simply had to conform to each existing standard.

The desire for an integrated language environment naturally results in shared code among VMS products. Initially, this sharing existed only at run time in a common math library and a common record management system. Eventually, however, engineers recognized that large sections of code were being duplicated over and over again in each compiler. Therefore, the next logical step was to develop a common compiler with language-specific sections to handle language-specific tasks such as parsing. On the VAX/VMS system, this common compiler is the VAX Code Generator (VCG).

At the time SCAN was being designed, the VCG supported the VAX PL/I and the VAX C compilers. Moreover, work was underway to support the VAX Ada compiler. The VCG included

- A global and peephole optimizer
- A comprehensive code generator
- Object module generation
- Listing and error-message facilities
- Command line interpretation
- Internal memory management
- Tools for debugging a compiler

During the design phase (Phase 1) of the VAX SCAN project, the team created a prototype for the product using the VCG to generate code for SCAN concepts that were new to the VCG. The success of this prototype effort showed that using the VCG concept was the correct solution to reducing the development cost of the VAX SCAN product.

The final results were impressive. By using the VCG, the team cut the cost of implementing the SCAN compiler by at least a factor of three. At the same time, the VAX SCAN compiler generates high-performance machine code to move strings and perform arithmetic operations that rivals the performance of any of the other VAX compilers.

The productivity benefits gained by using the common code generator are far reaching. Bug rates in the code generator are remarkably low, because the code generator is tested by four different languages. In addition, as new demands are made for compilers to support tools such as the VAX Language-Sensitive Editor, the work needed to support the tool is divided in two: the language-specific work, and the common work needed by all the compilers supported by the VCG. Frequently, this duality reduces the amount of work the VAX SCAN product must do to support such tools. Finally, each VCG improvement is actually an improvement for four languages. Thus, for example, a better register allocation algorithm benefits all four languages because it will be added to the common portion of each compiler.

The original vision of VAX/VMS products being well integrated had a handsome payback for the VAX SCAN product. First, it resulted in a set of standards that defined many of SCAN's external and internal interfaces. Second, integration fostered code sharing, which greatly reduced the cost of implementing the VAX SCAN software.

VAX/VMS Tool Set

Building software systems without the proper tools can be an inefficient and difficult process. Several of the tools available in the VAX/VMS environment are topics of other articles in this issue of the *Digital Technical Journal.*[4,6,7,8]

These tools, like the standards mentioned in the previous section, provided guidance to the VAX SCAN team by suggesting ways of solving problems. For example, the VAX Code Management System software describes a method for coordinating changes to the source code of the compiler and the run-time system. The VAX DEC/Test Manager software describes a means of testing the product. In general, evaluating and choosing a tool that exists is simpler and cheaper than designing one for the specific needs of a project.

A list of the major tools used during the development of the VAX SCAN product follows:

- The VAX Code Generator
- An LALR(1) Parsing System developed by the VAX Ada team
- The VAX/VMS Debugger
- The VAX RMS (Record Management System) software
- The VMS Run-time Library
- The VAX Language-Sensitive Editor
- The VMS Message Utility
- The VAX DEC/Test Manager software
- The VAX DEC/MMS (Module Management System) software
- The VAX DEC/CMS (Code Management System) software
- The VAX Performance and Coverage Analyzer
- The VAX BLISS Compiler and MACRO assembler

Of the three-developer-year effort required to implement the initial version of the VAX SCAN product, 8 developer-months were spent in the design of the language, compiler, and run-time system. The remaining 28 developer-months were spent implementing and testing the product. At the conclusion of the implementation phase, the system consisted of 75,000 lines of source: 60,000 in the compiler, and 15,000 in the run-time library. Forty-four percent of the lines were actual code, 20 percent were blank lines, and 36 percent were comments. The test system exercised over 90 percent of the code paths in the compiler.

Summary

The VAX SCAN project is regarded as a success at Digital for two rather diverse reasons. First, the product itself is a good balance of text processing features and integration with the VAX/VMS environment. The result for a VAX SCAN user is much greater programming productivity.

The second reason for success is attributable to the cost effectiveness of the project. The engineering costs of the product were low, the product was timely, and the quality was excellent.

The VAX SCAN product is a fine example of good ideas implemented by using sound software engineering principles and supported by effective tools.

References

1. A. Aho, R. Sethi, and J. Ullman, *Compilers, Principles, Techniques and Tools* (Reading: Addison-Wesley Publishing Co., 1986).
2. *Sperry Univac Series 1100 MACRO Programmer Reference* (Roseville: Sperry Univac Corporation, Issue: UP-8336 Revision 3).
3. *VAX SCAN V1.1 Documentation Kit* (Maynard: Digital Equipment Corporation, Order No. QL495-GZ-1.1, 1986).
4. B. Beander, "VAX/VMS Software Development Environment," *Digital Technical Journal* (February 1988, this issue): 10–19 .
5. *Software Development Policies and Procedures* (Maynard: Digital Equipment Corporation, 1981).
6. G. Lupton, "Language-Sensitive Editor," *Digital Technical Journal* (February 1988, this issue): 28–39 .
7. L. Ziman, M. Dickau, "Project Management of the VAX DEC/Test Manager Software Version 2.0," *Digital Technical Journal* (February 1988, this issue): 110–116 .
8. P. Gilbert, "Development of the VAX NOTES System," *Digital Technical Journal* (February 1988, this issue): 117–124.

Robert A. Conti

Software Productivity Features Provided by the Ada Language and the VAX Ada Compiler

The Ada language provides a number of features that can increase software development productivity. Many of these, such as packages, tasks, and exceptions are not present in conventional programming languages (such as C, FORTRAN, and Pascal). Others, such as strong typing rules, range-checking, and portability, are provided by some conventional languages, but not all. Beyond the inherent Ada language features, Digital's Ada compiler for the VMS operating system provides additional features that enhance productivity. Examples are automatic inlining, portability checking, and a comprehensive program library manager. This paper introduces the major productivity features of both the Ada language and Digital's Ada compiler, and describes some of the benefits that result.

Ada's Contribution to Productivity

For the purposes of this paper, software productivity will be defined to be the total profit generated by a software product divided by the total development costs (nowadays, mostly labor) required to design, develop, test, maintain, and enhance the product over its entire life cycle. This definition of software productivity is one that the manager of a commercial software business might use. By including both profit and expense, this definition also includes the effects of attributes that are associated with software quality, attributes such as

Compatibility

Ease of implementation

Ease of use

Execution time

Future extensibility

Maintainability

Memory space

Portability of applications

Reliability

Response time

Reuse of software components

Tailorability

Time to market

User expectations

It should be apparent that the traditional use of "lines of code per day" to define software productivity is incomplete in comparison with the definition used in this paper.

The U.S. Government developed the Ada programming language to help decrease the life-cycle costs of its computer programs. (Profit was not a factor.) Ada's features are intended to make software easier to design, read, and modify, as well as to be more reliable and portable between computer systems. In short, the features are intended either to reduce expenses or increase the quality of the software. Both of these effects make software development more productive according to our definition of productivity. Commercial enterprises should also be able to achieve improved software productivity by using Ada.

In March 1985, Digital released its VAX Ada compiler product for computers running the

VMS operating system. This product includes a number of additional features that reduce costs or improve software quality.

Although we have not made actual measurements of the productivity increases that would result from using Ada in general, and VAX Ada in particular, our experience in developing software in other languages indicates that certain features are quite beneficial. This paper introduces those features. We encourage the reader to reflect upon their collective impact.

Familiarity with a conventional programming language, such as FORTRAN, C, Pascal, or BLISS, is assumed. (References 1, 2, and 3 are textbooks containing detailed information on programming in the Ada language. Reference 4 is the more formal and technical language standard.)

Inherent Productivity Features of the Ada Language

The Ada language has inherent compiler-independent features that offer great promise for improving productivity. These features can be categorized as either "new" (those that would be novel to a FORTRAN, C, or even a Pascal programmer) or "improved" (those that may exist in other languages but have been improved in Ada).

New Features and Their Benefits

The new features of the Ada language support several modern software engineering concepts that can improve productivity.

Formal Separation of Specification and Body

Many software interfaces are designed with too much knowledge of the current implementation details. In some cases, the interfaces are even produced as a side-effect of doing the implementation. This common error of blurring the distinction between interface and implementation often results in haphazardly designed software with poor quality. The software is often difficult to extend (because it depends unnecessarily on some arbitrary detail of the first implementation), and its performance is hard to improve (because the implementation cannot be replaced by one with different arbitrary details). Moreover, the software is often too closely coupled to functionally unrelated software, is not portable to other targets, and is inconsistent in the view it presents to its clients. The term clients here refers to all the software that relies upon the features provided by the interface.

Good programmers have learned to informally discipline themselves to recognize the distinction between interface and implementation, and to design interfaces that are independent of an implementation. The Ada language formalizes this design practice by allowing a programmer to code the interface (the specification) separately from the implementation (the body) for program units such as procedures and functions. The language requires this separation for other program units such as packages and tasks (to be described later).

The following example illustrates this separation:[5]

```
-- The specification defines
-- what the caller can rely upon.
--
procedure SORT (X : in out WIDGETS);

-- The body contains the current
-- implementation.
--
procedure SORT (X : in out WIDGETS) is
    -- Declarations go here.
begin

    -- Details of the current sort
    -- algorithm go here.

end;
```

The following benefits accrue from formalizing this separation of the specification from the body:

- Improving the implementation is easier. Client software is less likely to depend on details of the first implementation; instead, it depends on just the interface.
- The client software is portable to other targets, because changes in implementation details are more likely to be hidden from clients rather than entangled in the interfaces.
- Consistent user views are maintained. The interfaces tend to be more logical because the design can be done before the implementation details are considered.
- The software tends to be more decoupled from unrelated software. (It tends to be more "modular.")

Packages

Programmers often use the term "software packages" informally to describe collections of related types, declarations, and operations. Conventional languages provide no way to bind such related software together.

The Ada language formalizes the concept of package. An Ada package has a specification, which contains what the client software sees, and a body, which contains the package's current implementation. The specification contains types, objects, and subprogram specifications that define the interface for the package's clients. The body supplies the bodies for the subprogram specifications and anything else needed to implement them. The specification and the body can be compiled separately, so that interfaces can be developed and checked long before the implementation is coded.

The following example shows a package specification and body for a window manager:

```
package WINDOW_MANAGER is

    -- This section would contain the
    -- specification of data,
    -- procedures, and functions that
    -- represent or operate upon
    -- windows, and can be used by
    -- client software.

    -- The specification for a
    -- procedure that creates
    -- windows.
    procedure CREATE_WINDOW
        (W : out WINDOW);
    ...
end;

package body WINDOW_MANAGER is

    -- This section corresponds to the
    -- current implementation of the
    -- window manager.

    -- The body of a procedure that
    -- creates windows.
    procedure CREATE_WINDOW
        (W : out WINDOW) is
    begin
    ...
    end;
end;
```

Packages promise easier use and modification of software. Related objects and operations tend to be grouped in the same package, thus making them easier for a user to find and comprehend. The collocation of related implementation software in the package body allows a maintainer to better understand the ramifications of a potential change.

Support for Abstract Data Types

Abstract data types are a relatively recent computer science concept. An abstract data type represents a set of abstract objects having a well-defined set of applicable operations. A client can operate upon an object only with the operations provided. Information about the internal structure of the object is hidden from the client.

The benefits of this information hiding are that the programmer of the client software is presented with a simpler conceptual model, one containing only the information that is essential for a client to know. Furthermore, client software cannot apply arbitrary operations to the object and thereby become dependent on accidental internal details that might need to change in a future implementation. Information hiding makes implementation of the object easier, because hiding clarifies which information current clients depend on and which information can be changed without affecting them.

Ada formalizes the concept of information hiding by allowing an implementor to declare "private types" in a package specification. Although clients can declare objects of a private type, only the implementation code in the package body can operate on the detailed internals of such objects. For example, if the object represents a window on the terminal, the client may only be allowed to request that it be shrunk, etc. However, the implementation may need to privately associate the window with a file and can store the file name within the window object, hidden from the client.

Figure 1 illustrates the two views of an abstract data type.

Ada's support of abstract data types should enhance an implementor's ability to extend applications in the future, as well as to make software more reliable.

Tasks

Most operating systems provide constructs to support concurrent execution, even on uniprocessors. But these constructs are typically very low level and difficult to use, and they are cer-

```
package WINDOW_MANAGER is
    -- The client knows the object only by the operations
    -- that are provided here.
    type WINDOW is private;
    procedure SHRINK_WINDOW (W : WINDOW; P : PERCENTAGE);
    ...
end;
package body WINDOW_MANAGER is
    -- The implementation knows the object by its internal
    -- structure.  In this case the window is really a record.
    type WINDOW_IMPLEMENTATION is
        record
            F : FILE_TYPE;
        end;
    ...
end;
```

Figure 1 Two Views of an Abstract Data Type

tainly not portable. The Ada language is one of the few widely available programming languages that has built-in constructs for concurrency (called tasks). Ada's tasks are both easy to use and portable; the language standard requires that all implementations support tasks in a specified manner.

An Ada task is like a procedure that executes in parallel with other parts of the program. A task starts executing as soon as the first statement of its declaring unit executes; tasks need not be started explicitly. Like packages and subprograms, a task has a specification and a body.

The following example illustrates the essential syntax of a task:

```
task T is
    -- This section would declare
    -- the entries that client
    -- software can call.
end;

task body T is
begin
    -- This section would contain the
    -- steps that the task
    -- executes in parallel with
    -- other tasks.

end;
```

Although not required by the Ada language specification, the Ada marketplace highly values the ability of an Ada compiler to execute other tasks while one task is waiting for an I/O completion. Implementations that support this feature (such as VAX Ada) provide a powerful reason to use tasks — even on a uniprocessor — to easily obtain greater throughput and responsiveness from an otherwise I/O bound program. On multiprocessors, users expect that an Ada implementation will assign different tasks to different processors for a program speedup. (VAX Ada does not currently have this feature.)

Using tasks written in Ada rather than the concurrency primitives provided by the operating system can lead to better productivity because

- Tasks are portable (less rework)
- Tasks are easier to understand (reduced maintenance)
- Tasks are easier to code (faster time to market)
- Using tasks on a uniprocessor results in free performance improvements when multiprocessor support becomes available

Like other Ada constructs, tasks also improve the overall thought process (even on a uniprocessor). Programmers soon stop "thinking serially." With tasks able to do asynchronous I/O, it becomes unthinkable to lock up the user's keyboard while a program does work that could be done in the background. As a result, products using tasks are more likely to be responsive to user inputs. This benefit is important for any product with a user interface.

Separation of Representation from Type

In many programming languages, integer variables are intimately bound to the attributes of a particular machine. For example, integer variables are often bound to the number of bits needed to represent some integer machine type.

The Ada language separates the inherent properties of a variable's type from the underlying machine types. For example, the declaration

```
type PLANET_NUMBER is range 1..9;
SPACECRAFT_LOCATION : PLANET_NUMBER;
```

says only that type PLANET_NUMBER needs a range of 1 to 9; the corresponding machine type is neither apparent nor important. The compiler, not the programmer, chooses which hardware data type will actually be used (for example, 8, 16, or 32 bits). This general concept — in which the most a programmer need specify are the range and precision, and the implementation chooses the detailed representation — is present for all Ada types. For cases in which detailed representations are important, the language provides a way to force data representations and storage layouts.

Separating the types from their representations achieves portability between machines (providing of, of course, that a machine type can be found that can satisfy the required range). Error checking is enhanced because the compiler automatically checks that the value of the variable stays within its declared range.

Exception Handling

In the conventional languages without built-in exception handling, a programmer must manage status variables that indicate whether or not a call to a procedure has failed. In addition, those languages provide no easy means for automatically passing error notifications to the calling routine or for cleanly specifying recovery actions.

In contrast, the Ada language has built-in features for handling exceptions. A programmer can decide which error conditions should be defined, when they should be signified, and how they are to be handled. Moreover, all these features are portable. Figure 2 shows how this feature works.

```
-- The error condition is declared.
LOST_THE_LINE : exception;
.
.
.
procedure GET_DATA is
begin
     .
     .
     .
     if STATUS /= NORMAL
     then
          raise LOST_THE_LINE; -- This statement signifies that the error has occurred.
     end if;
end;

procedure TEST_COMM_LINE is
begin

     -- This section calls routines that can raise exceptions.

     GET_DATA;

exception

     -- This section specifies all the error handling code for this procedure.

     when LOST_THE_LINE => NOTIFY_REPAIR_STATION;
                           raise; -- This statement passes the
                                  -- current error to the caller.
end;
```

Figure 2 Example of Ada's Features for Exception Handling

Because error handling is easily programmed in the Ada language, error-handling code is easier to read, and the constructed software is likely to be more reliable.

Dynamic Memory

Another feature often absent from conventional languages is the allocation and deallocation of dynamic memory. Although many operating systems provide this feature, most do not provide it in a portable form.

Ada, however, has a built-in dynamic memory feature that all Ada implementations support: pointer variables can be declared, and they can have dynamic objects assigned to them. Figure 3 describes how pointer variables can be used.

Ada's dynamic memory feature increases portability. In addition, because Ada's pointers are typed (can only point to objects of specified type), two other benefits accrue: It makes their use less error-prone, and it enables a compiler to exploit the additional knowledge it has about the objects, such as their maximum and minimum size, for optimization.

The Program Library

In most conventional languages, each procedure is compiled independently of other procedures. Thus it is possible to have a set of software routines that is inconsistent. For example, callers could assume certain conditions about the interface, yet those conditions may have changed.

The Ada language addresses this problem by requiring a program library to hold a consistent copy of all program units. The program library manager makes it impossible to link a program that has internal inconsistencies caused by obsolete interface specifications. This feature saves the time that would otherwise be spent tracking down obscure run-time errors.

The presence of a program library also creates the opportunity for Ada implementations to provide many other additional features and benefits. Some features that become possible, many of which are now common among implementations in the industry, are

- A simplified means of compiling and linking the entire program
- A simplified means of recompiling program units after another unit on which they depend has changed
- Query functions to answer questions about the program as a whole
- Ways of using multiple libraries together to realistically match project needs
- Optimizations that take into account more than one procedure (interprocedural optimizations)
- Subsystems, that is, the ability to restrict client software to use only certain allowed interfaces

The more important productivity benefits likely to accrue are faster development, and less time spent tracking down obscure errors.

Overloading

Most programming languages require a unique name for each program unit declared. The Ada language, however, allows any number of program units to have the same name, provided that the units have different interfaces, called signatures. This concept, called overloading, allows procedures that have the same logical effect but

```
type PTR is access NODE; -- This statement declares a pointer
                         -- type.
X : PTR;                 -- This statement declares a pointer
                         -- variable.
.
.
.
X := new NODE;           -- This statement dynamically
                         -- allocates an object and assigns
                         -- a pointer to it.
```

Figure 3 Ada Pointer Variable

that operate on different data types to have the same name. For example, a package defining both vectors and matrices could have two procedures called CREATE and two called DELETE. There is no need to construct artificially different names to differentiate between the procedures.

Overloading helps implement the principle of orthogonality, which means that each operation applying to one object type can also apply to any other object type, whenever that is meaningful. Overloading is simply a generalization of the common language feature allowing arithmetic operations to be applied to both integers and real numbers. Ada allows a programmer to define the meanings of most built-in operators for any programmer-defined data types; for example, one can define an addition operation for one's own matrix type.

Generics

Many languages force programmers to recode utility operations for each type of data on which they will operate. For example, different procedures are needed for sorting integers and one-dimensional arrays, even though a programmer may want to use the same sorting algorithm for both.

In contrast, the Ada language allows the definition of a generic form of package, procedure, or function. A generic program unit is independent of the type of data on which the program unit operates. This feature allows algorithms to be coded in their purest abstract form.

After defining a generic program unit, a programmer can then create an executable program unit by specifying the actual types upon which the generic unit is to operate. Creating an executable instance of a generic unit is called "instantiating" the generic unit. When instantiating the generic, one can also specify as parameters any procedures, functions, or tasks that need to be called by the unit. For example, a generic sort package may need to pass the function "less than" for the particular type to be sorted, such as a matrix.

Figure 4 illustrates the use of generics.

```
generic
      -- Declare that an arbitrary type is to be specified as
      -- the actual type to be sorted.
      type ELEMENT_TYPE is private;
      -- Since nothing is assumed about the type, the
      -- comparison function must be passed in as a parameter.
      with function "<"   (L: ELEMENT_TYPE, R : ELEMENT_TYPE)
           return BOOLEAN;
  package QUICKSORT is
      -- The sort package interface would be written here.
  end;
  package body QUICKSORT is
      -- The generic implementation would be written in this
      -- section; it would use the passed-in function "<" .
  end;
  -- The following statements create an instance of the QUICKSORT package
  -- for the type BOXCAR which is defined in package TRAINS.  The generated
  -- package is now capable of sorting BOXCARS using the quicksort method.
  with QUICKSORT;
  package QS_BOXCAR is new QUICKSORT (TRAINS.BOXCAR, TRAINS."<" );
```

Figure 4 Example Use of Generics in Ada

Generics allow the easy reuse of software, thus avoiding the inevitable errors caused by recoding an algorithm. The promise of generics is that one can truly code an algorithm just once.

Improved Features and Their Benefits

Ada improves upon a number of features that are commonly provided by other languages. In general, these features increase the amount of checking done at compile time, and thus save debugging time and make modifications easier to accomplish.

Strong Typing Rules

Like several other languages, Ada allows a programmer to define not just variables but types of variables. A type is simply a template for a set of values that share some properties. For example, the type named INTEGER describes a set of discrete, signed values with a certain range and permits certain well-defined operations, such as addition and subtraction. (A type does not have a size; objects have that property.)

The Ada language also allows a programmer to define a derived type, which is a new type whose properties are derived from a parent type. Although a derived type has the same properties as its parent type, its values, like the values of any other type, have different meanings and cannot be mixed freely with the values of other types.

Ada also provides the subtype construct, a way to name part of the full range of values of a type.

Ada's rules for mixing variables of different types within the same expression are stronger than the rules in other typed languages, such as Pascal. The only implicit type conversions allowed in Ada are between numeric literals and compatible numeric types. No implicit conversions are allowed between values of different types. Type conversions can be done, but programmers must explicitly write them. Every compiler is required to report an error if an attempt is made to combine different types without conversion.

The main benefit of these typing rules is that, when used properly, they help with error detection. An Ada compiler will detect many errors during compilation that might not even be checked for in other languages, or that might not immediately manifest themselves even at run-time. For example, typing can prevent the accidental truncation of a real number to an integer when rounding is required. Similarly, typing can prevent the accidental addition of a variable in feet to one in yards, or the addition of an employee's ID number to his or her salary.

Figure 5 illustrates several forms of type definition and use.

```
-- Define a machine-independent type and a subtype.
type PHYSICAL_NUMBER is digits 8 range  -1.0E35 .. 1.0E35;
subtype POSITIVE_PHYSICAL_NUMBER is
        PHYSICAL_NUMBER range 0.0..PHYSICAL_NUMBER'LAST;
-- Define new types derived from the above.
type FORCE is new PHYSICAL_NUMBER;
type MASS is new POSITIVE_PHYSICAL_NUMBER;
type ACCELERATION is new PHYSICAL_NUMBER;
-- Declare variables of each type.
F : FORCE;  M : MASS;  A : ACCELERATION;
-- FORCE and MASS cannot be mixed in the same expression
-- without an explicit conversion back to the base type,
-- PHYSICAL_NUMBER. Ada compilers must diagnose the following
-- illegal expression.
F := M;
-- The following statement is legal because there are
-- explicit conversions.
F := FORCE(PHYSICAL_NUMBER(M) * PHYSICAL_NUMBER(A));
```

Figure 5 Example of Forms of Type Definition

The productivity benefits of Ada's typing rules are reduced error tracking (and, hence, more productive use of development and maintenance time) and more readable and reliable programs.

Range Checking

In addition to providing strong typing rules, the Ada language requires that range checking be a default behavior (with most compilers permitting its suppression). Range checking involves either compile-time or run-time checks for computed values that exceed the legal range for a type.

In many languages, calculations can overflow without being detected. Not only are such errors often difficult to locate, but the erroneous results could cause an unpredictable and possibly disastrous outcome. For example, on one fateful day for a bank in New York, a number of real-time financial transactions overflowed the 16-bit variable; millions of dollars were lost as a consequence. If the program had been written in Ada, built-in checking could have detected the error the moment the first transaction overflowed.

Parameter-passing Checks

Many languages do not detect the passing of the wrong number of actual arguments (either more or fewer), or the accidental interchange of arguments with different types. These kinds of problems often lead to obscure run-time errors that are very difficult to track down.

Ada implementations, however, are required to detect and report such errors the moment the program unit is compiled. These checks immediately save debugging time and eliminate the deleterious effect that such errors have on product quality.

Validation and Portability

Ada, unlike many languages, has strict rules on what implementations are allowed to do. An implementation is allowed neither to extend the language it compiles nor to compile only a part of that language. As a result, all Ada compilers translate the same language, yielding increased portability between computer systems. Programmers need not learn the specifics of an implementation in addition to the language.

Before an Ada compiler can be called validated, it must pass a series of validation tests (currently over 4,000, but the number has increased with each release of the validation test suite). These validation tests not only ensure portability of Ada code but also force compilers to translate the whole language exactly as required. (A compiler must be validated once a year and must pass all new validation tests.) These tests ensure that legal programs are translated correctly and that all required error reporting, both compile-time and run-time, is accomplished.

Productivity Features Provided by VAX Ada

VAX Ada provides additional productivity features related to optimization, program library support, and smooth interaction with other VAX languages and the underlying VMS environment. (Detailed product documentation is provided in references 6, 7, and 8. Reference 9 describes how VAX Ada fits in with the VMS environment, and reference 10 describes some of the design decisions that were made in developing VAX Ada.)

Inlining

The VAX Ada compiler implements the language-defined pragma INLINE. (In Ada, a pragma is a compiler directive.) The compiler thus replaces calls to subprograms with inline code expansions (unless the subprogram uses a feature like tasks that prevent these expansions). In addition to honoring such explicit requests from the programmer, the compiler will automatically replace certain calls with inline code; the compiler uses a heuristic to decide if inlining results in more efficient execution than the call. (The heuristic considers both space and execution time.)

Inlining allows programs to run faster, and its use simplifies the conceptual design process. Call overhead is no longer a concern, and programmers can define the most logical interfaces.

Import/Export Pragmas

The VAX Ada compiler defines several pragmas to match the Ada language to the VAX Calling Standard in an optimum fashion. For example, the pragma IMPORT_PROCEDURE allows a call to a procedure written in any language. This pragma permits a programmer to specify parameter-passing mechanisms (reference, value, and descriptor), the procedure's external name, and other VAX-specific attributes that are not part of the Ada language.

These pragmas allow the mixing of Ada subprograms with existing programs written in other languages so that the benefits of Ada can be obtained for new code, and the benefits of reusing existing code can also be realized.

Portability Checks

VAX Ada can perform portability checks while compiling a program. These checks inform a programmer of the uses of potentially nonportable features. Such features include implementation-defined pragmas and other features that the Ada language permits to allow tailoring to the specific computer system. These checks allow a programmer to manage the trade-off between portability and access to implementation-dependent features.

Program Library Functions

As mentioned earlier, Ada's unique concept of a program library provides many benefits. The VAX Ada program library provides all of those benefits, as well as some others.

- Any program units made obsolete by revising a particular unit can be automatically recompiled.
- The compilation order for the program units of a program can be determined by simply naming the main program.
- An entire program can be built automatically by specifying just the name of the main program.
- Ada program units can be imported from other program libraries, or from other languages.

Asynchronous I/O Operations

The VAX Ada run-time library performs asynchronous input/output for all predefined Ada I/O operations. As a result, a programmer can use tasks to do computation and I/O in parallel and obtain greater throughput and more responsive user interfaces.

Support for Asynchronous Operations

The VMS operating system defines the asynchronous system trap (AST) construct for dealing with asynchronous events. An AST is really a software interrupt. Ada allows hardware interrupts to be mapped into a call to an entry point in a task. To support VMS software interrupts in a like manner, the VAX Ada compiler provides an implementation-defined attribute called AST_ENTRY. This attribute causes a software interrupt to generate a call to a task entry point. This feature simplifies the interfacing of Ada programs to the VMS environment.

Summary

The Ada features we have discussed make this language an excellent choice for general-purpose programming. Ada's principal productivity benefits are realized in software that is more extensible, portable, maintainable, reliable, and reusable. The VAX Ada compiler adds further features that enhance productivity. While each feature taken separately may not seem that significant, the combined benefit of all of them should be quite significantly improved software productivity (as defined earlier in terms of profit and life-cycle costs) relative to the use of other languages.

Acknowledgment

The author wishes to thank Barbara Rising Bishop for greatly improving the content and readability of this paper.

References

1. J. Barnes, *Programming in Ada* (Reading: Addison-Wesley Publishing Company, 1984).

2. G. Booch, *Software Components with Ada: Structures, Tools and Subsystems* (Menlo Park: The Benjamin/Cummings Publishing Company, Inc., 1987).

3. G. Booch, *Software Engineering with Ada* (Menlo Park: The Benjamin/Cummings Publishing Company, Inc., 1987).

4. *Ada Programming Language*, ANSI/MIL-STD-1815A-1983 (United States Department of Defense, U.S. Government Printing Office, 17 February 1983).

5. In the examples throughout this paper, lowercase is used to distinguish Ada reserved words from programmer-defined identifiers (shown in uppercase). Note that a comment in the language is identified by using a leading double hyphen (--).

6. *Developing Ada Programs on VAX/VMS* (Maynard: Digital Equipment Corporation, Order No. AA-EF86A-TE, 1985).

7. *VAX Ada Programmer's Run-Time Reference Manual* (Maynard: Digital Equipment Corporation, Order No. AA-EF88A-TE, 1985).

8. *VAX Ada Language Reference Manual* (Maynard: Digital Equipment Corporation, Order No. AA-EG29A-TE, 1985).

9. C. Mitchell, "Engineering VAX Ada for a Multi-Language Programming Environment," *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments, ACM SIGPLAN Notices*, vol. 22, no. 1 (January 1987): 49-58.

10. R. Conti, "Critical Run-Time Design Tradeoffs in an Ada Implementation," *Proceedings of the Joint Ada Conference, Fifth National Conference on Ada Technology and Washington Ada Symposium* (March 1987): 486–495.

Brian A. Axtell
William H. Clifford, Jr.
Jeffrey S. Saltz

Programmer Productivity Aspects of the VAX GKS and VAX PHIGS Products

The recent availability of high-level, device-independent standards for computer graphics programming has made the graphics programmer's task easier and far less time-consuming. Graphics programs, once major undertakings, can now be produced quickly and once written can be easily transported to other graphics devices and host systems. The VAX GKS and VAX PHIGS products are implementations of two of the major graphics standards. These products are based on a common architecture, the Base Graphics Architecture, consisting of five layers. The architecture was designed to incur minimum overhead in accessing high-performance devices, allow the reuse of many code modules, and provide easy extension of the products.

Computer Graphics Standards

Interactive computer graphics is the synergistic union of computer graphics output with an assortment of input techniques to facilitate operator feedback and control. Interactive graphics has long been recognized as the most effective available mechanism for man/machine interaction. The presentation of information in a visual form takes advantage of the superb pattern recognition capabilities of humans. Well designed input techniques enable much more natural and efficient interaction than is possible through keyboard input alone.

Throughout the early years of computing and well into the 1960s, the high cost of hardware and the even higher cost of software production inhibited the widespread use of interactive computer graphics. The expense of writing software was due in large part to the graphics interfaces of the period, which were typically device dependent and at a very low functional level. Because of the low level of these interfaces, the application had to do a lot of work to create even a simple image, thus making graphics programming difficult and time-consuming. The device-dependent nature of these interfaces often meant that programs had to be largely rewritten for each different device on which graphics output was to be produced.

By the mid-1970s, however, falling hardware prices and the availability of better proprietary graphics interfaces allowed computer graphics use to become fairly widespread. With this expansion came the realization that a standard graphics interface was needed that was both high level and device independent. Enough experience existed by then to attempt to design such a system.

The Core Graphics System, developed under the sponsorship of the Association for Computing Machinery, was the first significant device-independent graphics interface.[1] The concepts embodied in Core profoundly influenced subsequent graphics standards. Core never became an official standard, however, because too many nonconforming implementations became available before standardization could be achieved. Moreover, the experience gained from these early implementations of Core showed that a better interface could be designed.

The Graphical Kernel System, or GKS, is a beneficiary of the experience gained from Core and was adopted both as an ANSI and ISO standard in 1985.[2] It is the first official ANSI/ISO graphics standard. Digital's implementation of GKS for the VMS operating system, called VAX GKS, has been available since 1984 and is now in its third release.

The Programmer's Hierarchical Interactive Graphics System, or PHIGS, is being developed as both an ANSI and an ISO standard.[3] PHIGS has a different architecture and different design goals from GKS; it is compatible with GKS, however, wherever this compatibility is technically feasible. The first release of VAX PHIGS became available in January 1988.

The Role of VAX GKS and VAX PHIGS Products in Software Productivity

The VAX GKS and VAX PHIGS products are high-level functional interfaces that make graphics programming easier and more straightforward. A competent programmer should be able to develop graphics software after a short period of study. Moreover, the use of these systems eases the difficulty of finding trained graphics applications programmers. Many colleges and universities teach GKS as part of their computer-graphics curricula; there is evidence that they will teach PHIGS as well.

Each standard is built on the model of an abstract graphics workstation with standardized input and output capabilities. That is, each standard defines an idealized device for acquiring input from the operator and for generating graphics output. To a large extent, the job of the GKS or PHIGS implementor is to provide a layer that makes a particular real device behave like an ideal device. We call such a layer a "workstation handler." If the abstract workstation model of GKS or PHIGS can be implemented quite directly on a real device, such a device is called GKS-like or PHIGS-like.

This device-independent approach means that a program written to the GKS or PHIGS interface can, within limits, run unchanged on any supported device. Moreover, if an application is written to one of the standard language bindings (i.e., uses standard function names and parameterization), then the application can easily be ported to any other GKS or PHIGS host system supporting that language binding.

Although the basic functionality of both GKS and PHIGS is strictly defined, Digital's implementations further increase programmer productivity in several ways.

- Both VAX GKS and VAX PHIGS provide for very easy support of additional devices. Any user can add support for a new device by simply writing a few functions and building a table defining the capabilities of the device. (Refer to the Workstation Manager section for more details.)
- VAX GKS or VAX PHIGS workstation handlers are activated at run time, not directly linked with the application. Therefore an application can operate any supported device without relinking, thus minimizing the link time and executable program size of an application. This allows the application programmer to spend more time doing productive debugging.
- Both VAX GKS and VAX PHIGS provide extended input and output functionality. Additional output primitives implement common objects (for example, circles and ellipses in VAX GKS), thus saving the programmer from having to build them from lower level primitives. Extended output functionality (for example, lighting, shading, and depth cueing in VAX PHIGS) provides services not easily layered on top of the basic model.
- Both VAX GKS and VAX PHIGS integrate cleanly into a windowing environment. VAX GKS and VAX PHIGS enable the creation of applications with the same "look and feel" as other applications written using low-level windowing commands. The application programmer is thus freed from concerns about the windowing system, and the user sees a consistent windowing interface.
- Support for many devices is available through both VAX GKS and VAX PHIGS. VAX GKS is supported on VWS workstations[4] and on ReGIS, HPGL, POSTSCRIPT, and Sixel devices. PHIGS is supported on these devices as well as on the VAXstation 8000 high-performance workstation.

VAX GKS and VAX PHIGS as Complementary Graphics Standards

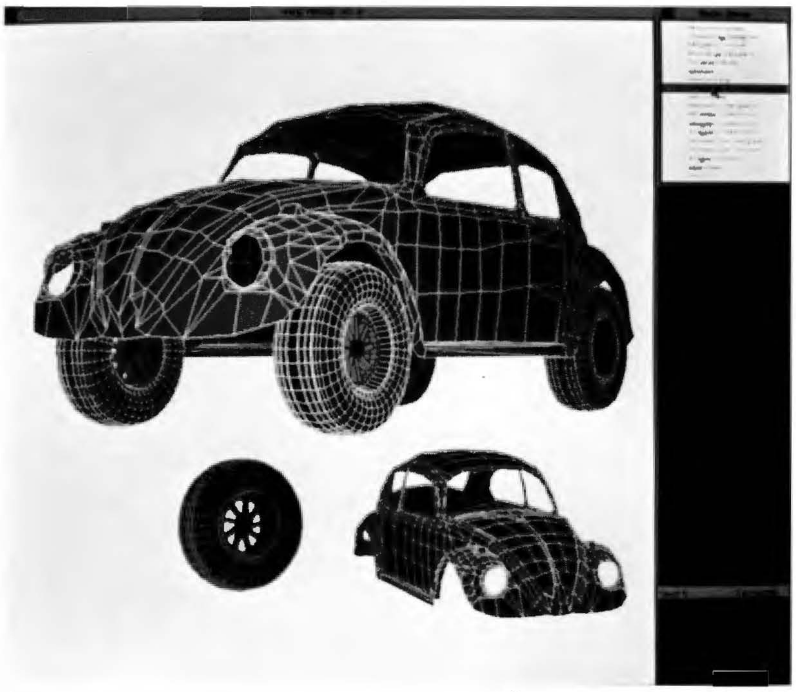
GKS and PHIGS are designed to satisfy different needs. GKS is a two-dimensional interactive viewing system. That is, GKS provides a mechanism by which images, described as collections of two-dimensional output primitives, can be displayed. In addition, GKS provides a variety of operator input methods. Although GKS can be used to display graphs and charts and other two-dimensional information, it can also be used as the imaging stage of a higher level system. Such a system first does the necessary processing to convert its objects (for example, objects defined in three-dimensional space) into GKS two-dimensional output primitives for display.

GKS commands for setting attributes of primitives and for generating output primitives can be either aggregated into collections called "segments" or executed immediately without being retained in a segment. The set of defined segments can be used later to rebuild the image on a display, thus the application does not need to reissue those commands. GKS segments are not revisable; once defined, the contents of a segment cannot be modified. However, each segment has several attributes (for example, a segment transformation and a segment visibility control) that can be changed.

PHIGS is a three-dimensional interactive modeling system. Like GKS, PHIGS offers operator input techniques; unlike GKS, however, PHIGS primitives are defined in a three-dimensional coordinate system rather than a two-dimensional system. PHIGS also does not have immediate execution of output commands. Instead, all commands that set attributes of output primitives, generate output primitives, or define other aspects of the graphics database are aggregated into revisable collections called "structures." This approach is motivated by two of the goals of PHIGS: to be able to describe and implement the application model (the concept or system being modeled by the application) and to allow the efficient revision of that implementation and its corresponding image.

Since typical application models are multilevel, PHIGS structures can be organized hierarchically. The relationship between levels of the hierarchy may represent geometric relationships (for example, the positional relationship between the components of an articulated arm) or other logical relationships as dictated by application requirements. (One can think of GKS segments as providing a single level of hierarchy.)

In PHIGS, for example, one can define a "wheel" structure (a collection of commands that describes something having the appearance of a wheel) and a "body" structure. One can then define a "car" structure as an assemblage consisting of four instances (displayed images) of the wheel and one instance of the body structure. Each instance has a geometric relationship to the car (that is, it has some position with respect to the car). The car can be moved around as a unit, with the relative positions of the wheels and body maintained by virtue of the geometric relationships imposed on them. This concept is illustrated in Figure 1.



Wheel and tire data courtesy of Brigham Young University. Car body data copyright © Evans and Sutherland Computer Corporation.

Figure 1 PHIGS Structure Hierarchy

GKS and PHIGS share a common set of output primitives, although GKS allows their definition only in two dimensions, whereas PHIGS allows it in three. These primitives are lines, symbols ("markers"), polygons, text, and pixel arrays. GKS and PHIGS also share a common input model. This model defines six input device types that are abstractions of common physical input devices, such as pointing devices, dials, button boxes, and keyboards.

GKS was designed as a very general interface on top of which many different classes of applications can be built. In fact, it is possible to build a layer on top of GKS that emulates PHIGS. PHIGS is also a very general interface, and the range of applications that can make use of PHIGS' modeling and three-dimensional capabilities is large and diverse. However, PHIGS is a bit more specialized than GKS, and building some styles of graphics applications on top of PHIGS is difficult if the application model cannot be made to fit PHIGS' modeling facilities.

The benefits of incorporating modeling into PHIGS are twofold. First, the application writer has a well developed, standard modeling facility available. Second, it becomes economically feasible for hardware vendors to provide direct hardware support for PHIGS modeling because the modeling facility is standardized. Such devices can provide very high performance.

The VAX PHIGS and VAX GKS products have a common architecture called the Base Graphics Architecture, described next.

Base Graphics Architecture

Digital's Base Graphics Architecture was designed as a general framework for implementing graphics systems. It takes a layered approach and consists of five components:

- Application layer
- Language binding layer
- Kernel layer
- Workstation handler layer
- Device layer

This approach is shown in Figure 2. Following a discussion of the design goals of the Base Graphics Architecture, each of these components is described.

Design Goals

The design team set a number of goals for the design of the Base Graphics Architecture.

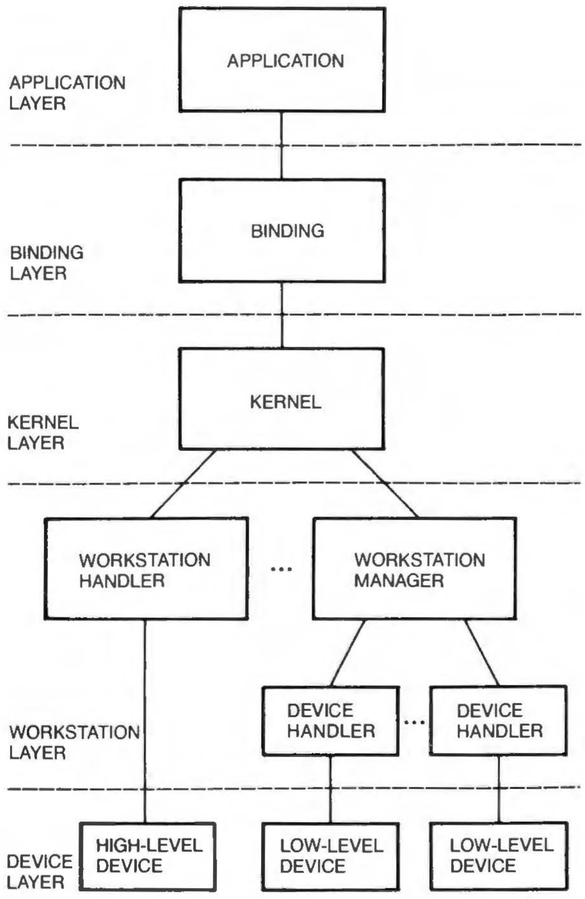


Figure 2 Layers of the Base Graphics Architecture

- VAX GKS and VAX PHIGS products had to conform to their respective standards.
- Since performance is critical in graphics systems, the architecture had to allow access to any high-performance hardware features of a device. Moreover, the system had to incur minimal overhead in using those performance features.
- Adding support for new devices had to be relatively easy.
- Wherever possible, components of each implementation of the architecture had to be interchangeable and reusable.
- The architecture had to provide a mechanism for customers and third-party vendors to write graphics handlers for their own devices and to integrate them into the system.

- The architecture had to be extensible enough to allow for changes in both graphics hardware technology and graphics standards.

Each layer of the Base Graphics Architecture is described below.

The Application Layer

The application layer is not a part of the graphics system but is, rather, a user of its services. The application programmer defines and has total control of the application layer. In reality, the application layer is typically another series of layers of increasing functionality, not the monolithic component depicted in Figure 2. The interface seen by the application layer is called a language binding and is supported by the binding layer.

The Binding Layer

A language binding is a functional interface to the capabilities of the graphics systems. A binding layer consists of several bindings, but a typical application uses only one language binding. Each binding is oriented toward some particular language or calling convention. The language bindings of the binding layer fall into two categories:

- Standard bindings for single languages. These are language bindings developed by the same organizations that developed the graphics standards. For example, the VAX GKS binding layer includes a FORTRAN language binding that conforms to an ANSI/ISO standard. Such a single-language binding is designed to be consistent with the capabilities of the particular language. Programs written to such standardized bindings are portable to any implementation of the same binding for that graphics standard.

- VAX calling convention bindings. Each binding layer provides a VAX run-time library (RTL) binding, called the GKS$ and PHIGS$ bindings for VAX GKS and VAX PHIGS, respectively. These bindings conform not to international standards but rather to the VAX/VMS calling standard. Therefore, VAX GKS and VAX PHIGS can be accessed through these bindings from any language that conforms to this calling standard.

The language binding layer is typically a very thin shell over the kernel layer.

The Kernel Layer

The kernel is the portion of the architecture that manages and controls the device-independent operations of the graphics system. Its main function is to act as a router, directing commands to the appropriate workstation (multiple workstations can be simultaneously active in both GKS and PHIGS) and to serve as a collection point for input events generated by the input devices. The kernel also maintains all information about the state of the system as a whole and is capable of responding to inquiries about system state and facilities. For example, an application can inquire about the types of devices available or the number of active workstations. Furthermore, the kernel is responsible for reporting errors back to the application.

Another responsibility of the kernel is to activate the workstation handlers. These components of the workstation layer are not linked directly with the higher levels of the system, but instead are built as shareable images. When the services of a handler are first needed, the kernel activates the handler through a VMS library routine.

The advantages of dynamically activating a workstation handler, rather than linking some or all handlers directly with the application, are as follows:

- A user-supplied handler can be incorporated without the need to link it (that is, using the VAX linker) directly with the application and kernel. It is only necessary to define several logical names that indicate the file name and entry-point table symbol name for the particular workstation type. (An entry point table is a structure similar to a VMS transfer vector.)

- Link time is substantially reduced because an application is only linked against the language binding interface, which is itself a shareable image.

- The amounts required by the application of both disk space and virtual address space are significantly reduced.

The VAX GKS and VAX PHIGS kernels are optimized for the most common functions. They incorporate various caching schemes and "hot paths" to accelerate performance for expected configurations and call sequences. Therefore, for many functions, the kernel merely has to perform one or two tests and then call the next layer.

The Workstation Layer

The collection of workstation handlers that constitutes the workstation layer is responsible for implementing the workstation abstraction of its particular graphics standard. For any graphics system based on the Base Graphics Architecture, the workstation handler interface must be defined at a very high level to allow access to the high-performance features of a device.

The functions of the workstation handler interface for VAX GKS, for example, are basically one-to-one with GKS functions that deal with the input, output, and workstation state. The workstation handler interface for VAX PHIGS has a similar relation to the PHIGS functions. Thus the implementor of a workstation handler can take advantage of the capabilities of those devices that closely match the workstation abstraction. Although the workstation handler interface thus includes many entry points, the implementation of each function should be relatively straightforward for devices that closely match the workstation abstraction.

However, most current devices do not have an architecture that closely matches the workstation handler interface. Very few devices, for example, could be considered GKS-like or PHIGS-like (though this situation is slowly changing). The job of writing a workstation handler for a low-level device is indeed an arduous one. To minimize the effort needed to interface such a device, an abstract graphics device at a much lower level has been defined. Therefore, each implementation of the Base Graphics Architecture needs a workstation handler that implements that implementation's high-level workstation abstraction for this low-level abstract graphics device.

VAX GKS, for example, has such a special workstation handler that implements the GKS workstation abstraction for the low-level device. VAX PHIGS also has one for the PHIGS abstraction. This special workstation handler is called a workstation manager; an implementation of the low-level abstraction is called a device handler, as shown in Figure 2.

Workstation Manager

To the kernel, the workstation manager is just another workstation handler. It is activated the same way and is accessed through the same interface as other workstation handlers. After activating the workstation manager image, the kernel calls the "open workstation" function of the workstation manager. The workstation manager, in turn, activates the appropriate device-handler shareable image, again through a VMS library routine.

The main job of the workstation manager is to span the semantic distance between the workstation handler and the device handler interfaces. The exact nature of that job differs depending on the abstract workstation implemented by the workstation manager. However, a typical workstation manager does the following tasks:

- Maintains state information on behalf of the low-level device
- Performs necessary geometric transformations
- Simulates functionality not available through a particular device handler
- Performs data management of aggregated output primitives and attributes (for example, GKS segments or PHIGS structures)
- Responds to inquiries about workstation state and available facilities

In reality, the design of the device handler interface requires that each workstation manager implement its particular workstation abstraction for a range of abstract low-level devices. That is, a device handler need not implement the entire low-level abstraction. The workstation manager is expected to simulate those functions not supplied by the device handler. In fact, the only mandatory output function for the device handler is a function which draws a series of connected lines. If necessary, the workstation manager will simulate all other output primitives in terms of that single primitive. Similarly, most of the input functionality of the device handler is optional; the workstation manager will simulate the missing functionality.

For example, both VAX GKS and VAX PHIGS can generate polygons whose interiors can be filled with a solid color or with various crosshatched patterns. The implementor of a device handler may choose to support this primitive directly if the device for which the device handler is being written has this capability. If the device does not provide this capability, the device handler can have the workstation manager simulate filled polygons using the line-drawing primitive of the device handler.

Device Handler

A single device handler interface is common to all implementations of the Base Graphics Architecture. As a result, PHIGS supported several dozen devices immediately when the development of the PHIGS workstation manager was complete. Those supported devices were the ones for which device handlers had previously been developed in support of VAX GKS.

The device handler interface defines 27 functions, a device description table, and an entry-point table (a transfer vector). The workstation manager consults both the device description table and the entry-point table to determine what functionality is available through the device handler and what must be simulated.

Because of the adaptability of the workstation manager, a new device can be added by writing just seven device handler functions and then building the device description and entry-point tables. Such a minimal implementation will not provide optimal performance for most devices, but will allow them to be put into service quickly. Over time, the implementor of a device handler can add more device handler functions to take advantage of the capabilities of the device. As it becomes available, each new version of the handler is placed into service simply by installing it in place of the previous version. Relinking the application program, kernel, and workstation manager is not required.

Device Layer

The device layer, the lowest layer in our architecture, marks the lower boundary of the Base Graphics Architecture. This layer consists of the various devices made available to the application by the higher levels of the architecture. The interface to this layer is device dependent.

Integration with Windowing Systems

The Base Graphics Architecture was designed so that a windowing system could be treated as just another device type within the Base Graphics Architecture. The model for supporting windowing systems realizes each instance of a "workstation" (the abstract device, not a specific device, such as a VAXstation II/GPX workstation) as a separate window on the device's display. Thus multiple GKS and PHIGS workstations can be active on the same device under the control of one or more applications. For example, "workstation" windows for both PHIGS and GKS and a third window created through the graphics commands of the native windowing system can all coexist on the same display, as shown in Figure 3.

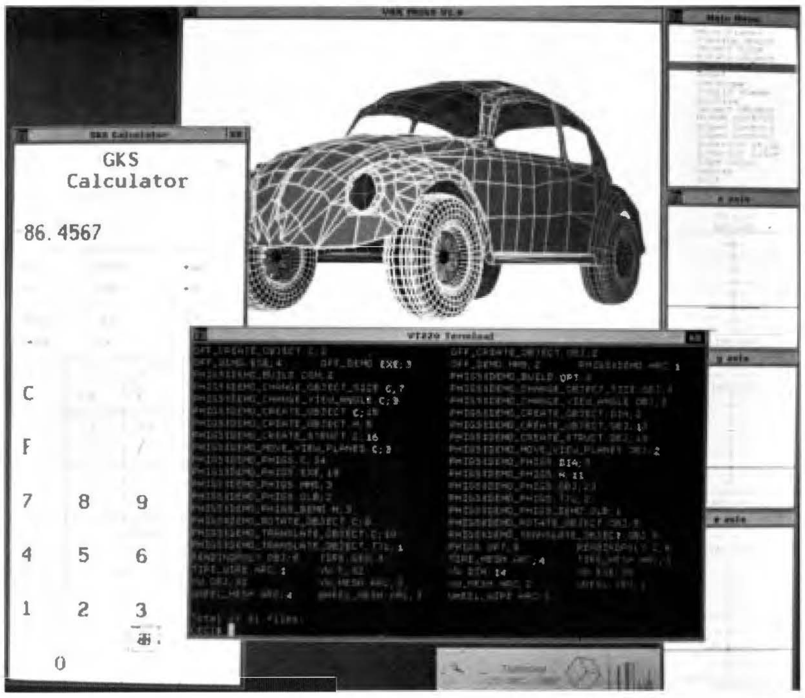


Figure 3 Three Workstation Windows in One Screen Display

The method by which a windowing system is supported (whether through a workstation handler or a device handler) logically depends upon the level of graphics support provided by the windowing system. For example, the VMS Windowing System (VWS) is supported through a device handler because VWS is neither PHIGS-like nor GKS-like enough to warrant writing a full workstation handler.[4] However, if three-dimensional and other higher level capabilities exist in a windowing system (for example, the proposed X3D-PEX extension[5] to the X Window System[6]), then it might best be supported with a workstation handler.

The implementor of the handler can use any tool kit that the windowing system provides to create windows and to perform certain classes of input operations. For example, typical tool kits provide menuing capabilities that can be used to support the CHOICE input type defined by PHIGS and GKS. When the tool kit is used for all possible windowing operations, all windows have the same appearance to the user or application programmer, even when they are generated by different graphics standards. The net effect is a graphics standard operating within the windowing environment.

Extensibility in the Base Graphics Architecture

The Base Graphics Architecture includes a mechanism that allows an implementation to provide extensions in a manner conforming to standards. Such extensions can define additional output primitives and provide extended control capabilities. The VAX PHIGS and VAX GKS products use these mechanisms to provide extensions supported by Digital. In addition, the design of the Base Graphics Architecture enables the implementors of workstation or device handlers to add their own extensions. The special capabilities of a particular device, not otherwise accessible through the standard functionality, can be made available in this way.

VAX GKS Extensions

VAX GKS has extended output primitives for generating various unfilled and filled circles, circular arcs, ellipses, elliptical arcs, and rectangles. These are frequently needed primitives which an application programmmer would otherwise have to generate using standard GKS primitives. The implementation of these primitives as extensions can also take advantage of any support provided by the underlying device (for example, if the device has a circle primitive). VAX GKS also provides control over such things as line join and cap styles, as well as primitive "writing modes" (for example, replace, complement, negate).

VAX PHIGS Extensions

In November 1986, an ad hoc working group representing some twenty companies and universities, including Digital, was formed to propose and develop extensions to PHIGS in the area of lighting, shading, depth cueing, back-face surface processing, and curve and surface representation. The set of functionality formulated by this group is called PHIGS+.

While PHIGS+ is not an official standards effort, a baseline document has been made available to the members of the ANSI PHIGS committee for comment.[7] It is the intent of the ad hoc PHIGS+ working group that a revised PHIGS+ draft be made available to the official standards bodies when the document is complete.

VAX PHIGS includes extensions in most of the areas being addressed by the PHIGS+ group. VAX PHIGS supports depth cueing, back-face surface processing, several different types of lights, various surface rendering effects (methods for simulating shiny or matte surfaces), and an advanced output primitive.

These extensions are defined in the VAX PHIGS workstation handler interface. Where possible, these extensions are also simulated by the PHIGS workstation manager. Therefore, within the limits of a particular device, these extensions are available on all devices supported through the device handler interface. Using these extensions effectively, however, is possible only on a device that can simultaneously display a reasonably large number of colors or shades of gray. Currently the PHIGS workstation manager requires that a device be able to simultaneously display 64 colors in order to simulate these extensions.

Summary

The VAX GKS and VAX PHIGS products are extended implementations of the existing GKS and proposed PHIGS computer graphics standards, both of which are high level and device independent. Both PHIGS and GKS make computer graphics programming far less complex than in the past. Moreover, they allow program portability among different graphics devices and different host systems. These qualities can lead to greatly increased application programmer productivity.

Both VAX GKS and VAX PHIGS are based on a single architecture designed by Digital. This architecture allows the efficient utilization of high-performance devices, the reuse of large portions of code during implementation, flexibility in the approach taken to support a particular device, and access to the unique capabilities of a device. This approach has boosted the productivity of the VAX GKS and VAX PHIGS implementation teams, and is expected to minimize the work required of third parties to add device support.

Acknowledgments

The authors wish to acknowledge the tremendous contributions made by the VAX GKS and VAX PHIGS development teams: Dwight Brown, George Chaltas, Kenney Chan, Keith Comeford, Glenn Davison, Jeff Ford, Garry Poegel, and Moh-Fung Shen. This effort could not have been made without continuing guidance from John McConnell. Finally, we wish to thank Joy Kinnear for her constant support of our efforts.

References

1. *Computer Graphics,* vol. 13, no. 3 (August 1979).

2. Standard ISO 7942 for Information Processing Systems, "Graphical Kernel System (GKS)," International Standards Organization (1985).

3. Draft Standard ISO 9592 for Information Processing Systems, "Programmers Hierarchical Interactive Graphics System (PHIGS)," International Standards Organization (1987).

4. *MicroVMS Workstation Graphics Programming Guide* (Maynard: Digital Equipment Corporation, Order No. AA-G110B-TN, 1985).

5. *PEX Protocol Specification, Version 3.00* (Boston: Massachusetts Institute of Technology, 1987).

6. *X Window System Protocol, Version 11, Project Athena* (Boston: Massachusetts Institute of Technology, 1987).

7. PHIGS+ Functional Description, Revision 2.0, report issued by the ad hoc PHIGS+ Committee (1987).

Lewis Lasher

The VAX RALLY System — A Relational Fourth-generation Language

The VAX RALLY system, a forms-based fourth-generation language, is designed to simplify the production of interactive database applications. The designers of this system sought a balance between ease of use and flexibility in the development of the object-based definition system. The definition system allows commonly anticipated features to be implemented by nonprocedural means, and other features to be implemented by means of escapes to other languages. The run-time environment allows many users non-interfering, concurrent read/write access to the same data. The representation of an application by a set of objects allowed the definition system to be implemented as a RALLY application. This use of RALLY for its own user interface gave the designers a fast and effective means to make product improvements.

The VAX RALLY system is a forms-based, fourth-generation language (4GL), or application generator, for database applications. Like other fourth-generation languages, it increases the productivity of application definers by providing them with high-level constructs that simplify application development.

This paper discusses the design principles that have contributed to RALLY's usability by application definers and to the efficient development of the RALLY definition system itself. In particular, attention is given to the design trade-offs inherent in 4GLs; the design of an object-based environment; and the use of the same data model for application data as for the application itself.

Product Design Overview

Version 1.0 of the VAX RALLY system was developed over a three-year period (1983–1986) as a combined effort by Digital Equipment Corporation and Foundation Computer Systems, Inc., the original authors of the "ALLY" 4GL product. Digital is currently developing later versions of the VAX RALLY system.

VAX RALLY is an application definition system and run-time environment for applications using Rdb/VMS databases. The definition system consists primarily of the following:

- An object-based, nonprocedural set of tools
- A procedural language whose syntax is adapted from the Pascal language
- Interfaces to programs written in traditional, third-generation programming languages (3GL)

The run-time environment provides the following:

- Virtual multitasking in a single VMS process
- Flow control within and among tasks
- Screen painting
- Capture and validation of input from the user of an application
- Data manipulation operations to and from databases

The users of VAX RALLY fall into two classes corresponding to its two major divisions: application definers who use the definition system to build applications, and end users who use these applications. However, because the definition system itself uses a RALLY application for its forms and menus, both classes of users, and indeed both of the major divisions of RALLY, are affected by the same design decisions.

Trade-offs between Ease of Use and Power

A recurring trade-off in the design of RALLY, as in 4GLs generally, involves the tension between the goals of ease of use and power, or flexibility. If the 4GL is too complicated, then users may find it simpler to continue using traditional 3GL programming methods. If the 4GL is not sufficiently powerful to meet the users' needs, users may have no choice but to resort to 3GL programming.

RALLY accommodates both these goals in several ways. First, RALLY is specifically optimized for the development and execution of a particular class, albeit a large class, of applications, namely, interactive database applications. Second, RALLY provides the application definer with a small set of carefully chosen objects and the ability to combine these simple objects into complex combinations. Finally, RALLY provides four different, partially overlapping approaches to application development: the builder tools, the editing environment, an integrated procedural language, and the ability to interface with programs written in traditional programming languages. In effect, the designers solved the problem of having to trade off either ease of use or flexibility by resolving the problem in, not one, but several aspects of the RALLY software.

The builder tools are a greatly simplified subset of the features available in RALLY as a whole. Few choices are given to the application definer, extensive defaulting is used, and many RALLY concepts are simplified or omitted altogether. By using the builder tools, a novice can learn the most important RALLY concepts immediately, build a usable application in a short time, and defer learning other RALLY concepts until needed.

The editing environment is the largest part of RALLY. It is almost entirely forms-based. An application definer fills in blanks on forms to specify the options to employ or the connections to make between RALLY objects. Using these forms, the definer has no need to learn or recall a language syntax. Although the set of features RALLY offers is not simple, the entire set is organized through RALLY's menus, and each form is labeled. Because each form presents a manageable amount of functionality, it can be documented using on-line interactive help messages specific to each form or to each field on a form.

RALLY includes an integrated procedural language called ADL (Application Development Language). ADL can be used for application features, such as arithmetic formulae, that are easier to describe using a language than by filling out forms. A definer typically uses an ADL procedure to specify the formula for a computed field, to define the conditions for validating data operations, to manipulate data outside of a form or report, or to alter flow control.

Lastly, a RALLY application can be integrated with programs written in traditional programming languages. Although this does not contribute directly to the productivity improvements realizable by using a 4GL, it expands the range of applications that can use RALLY.

Kinds of Applications Generated

The word "application" can refer to practically any effect achievable by traditional, 3GL programming languages. The goal of RALLY is more focused: to simplify the production of interactive database applications. Within the narrower domain of these data processing applications that RALLY generates, it is possible to predict what features are most likely to be useful. The conceptual basis for both the definition system and the run-time system was largely governed by such predictions about the types of applications RALLY would generate.

The VAX RALLY product, like other 4GLs, simplifies application development by providing the application definer with a small set of high-level constructs and the tools with which to combine them. Because RALLY focuses on database applications, many of RALLY's constructs are database operations: reading, inserting, deleting, and updating records; and committing and rolling back transactions. The major components of such applications are forms through which end users enter data to be written to the database, and reports on which end users see data that has been read from the database. Finally, although the primary focus of RALLY is on interactive applications, RALLY is also designed to be able to process records in batch.

The Object-based Definition System

RALLY conceives of an application as an interconnected network of objects. Each object has attributes that represent characteristics of the application.

Virtually an entire VAX RALLY application can be defined by filling out forms to specify the attributes of and connections between objects. The application definer gives each object a name that is used when connecting it to another object. Because the application definition system controls the storage and retrieval of objects, it always makes available to the application definer a list of all the objects at the definer's disposal. These lists, called lists of values, improve productivity because the definer need not rely on memory or written notes. Because the definer can specify connections by moving the cursor to the list of values and pointing at the appropriate name, without having to type the name manually, the definer is encouraged to give objects long, descriptive names.

The major types of objects that may be defined in a RALLY application are tasks, menus, form/reports, data source definitions, ADL procedures, external program links, number formats, and date formats. Form/reports and data source definitions, which are discussed in more detail later in this paper, contain subobjects such as fields.

The design of these objects was governed by principles of modularity. When a set of characteristics is likely to be used together, those characteristics belong in the same object. When a subset of these characteristics is likely to be changed while the other characteristics remain unchanged, that subset may belong in a separate object. These decisions are heavily dependent upon knowledge of how the characteristics are viewed and used by users, in this case, by application definers.

The most significant design decisions in the set of RALLY object types were

- The separation of data source definitions from forms and reports
- The unification of forms and reports into a single object type

The following sections describe the data source definition and form/report objects, and the data groups which are the basic structure of form/reports.

Data Source Definitions

The data source definition (DSD), unlike the form/report or menu, is not an object with an obvious justification for existence. Forms, reports, and menus are visible components of applications; DSDs are invisible auxiliaries to forms or reports. Yet there are two strong reasons for the existence of the DSD as a separate object: data independence, and reusability.

Data independence means the isolation of an application from unnecessary dependence upon the details of data storage. Most characteristics of a database application, and most characteristics of a form or report, are unconcerned with such details. RALLY isolates these storage details, such as the type of database and the names of relations and databases, in the DSD object. The form/report object handles the user-visible features of the application, such as how data is formatted and, in the case of forms, validated.

Reusability refers to the ability to make several uses out of information that is defined only once, thus avoiding redundant and time-consuming work. The same DSD, describing the same source of data, may connect to several different forms and reports. Conversely, a form/report can be reconnected at different times to different DSDs, allowing a form or report with the same features to operate on a different set of data.

An additional reason to separate out DSDs as objects in their own right is the design goal of supporting both interactive and batch processing. Interactive data processing occurs in RALLY in form/reports; batch processing occurs in ADL procedures. For both types of data processing, an application definer must specify the source of data, restrictions on record selection, and a locking strategy. The information that must be specified for both interactive and batch processing is contained in the common object, the DSD.

The existence of DSDs as separate objects contributes to the goal of simplifying ADL syntax. A small set of high-level primitive functions serves for all access methods.

Form and Report Functions in a Single Object

The VAX RALLY system treats both forms and reports as a single object, called a form/report. Despite the common practice, even in this paper, of referring to forms and reports as distinct phenomena, they share essential characteristics for the display and formatting of data records with accompanying text. Providing a single construct simplifies the concepts that a definer must learn. Moreover the form/report is an inclusive, not disjunctive, generalization of the characteristics of both forms and reports. Not only the conceptual definition of the form/report object but also each instance can include the union of the sets of form characteristics and report characteristics. Because a single RALLY form/report object can handle the entire set of interactive data operations, the definer can attain considerable productivity after mastering a small set of concepts.

A form/report by default reads records from one or more data sources, displays them to the user, and performs data manipulation operations in accordance with the end user's actions, writing data out to the data source(s). By default, an end user may browse through the records displayed, modify or delete them, insert new records, commit or roll back the database transactions, perform queries to view a subset of the data, and perform these same data operations on the subset shown as a result of the query.

Each capability can be removed outright from a given form/report or restricted conditionally. For example, to make a form/report behave like a traditional data entry form, the definer may eliminate the capabilities of reading existing records and querying.

Data Groups — Building Blocks of RALLY Form/report Structure

Traditional forms processing software confines its function to the collection of data, leaving the programmer to write the collected data to a file or database. The RALLY designers recognized that, once collected, data is most commonly written out to a data source. Consequently RALLY provides as a standard option the combined functionality of collecting data and writing it to a file. RALLY also allows for "pure forms" that do not automatically write out data.

The most significant design feature within RALLY form/reports is the data group. This structure is specifically designed for forms or reports connected to a data source, such as a database or a file with normalized data.

The data group itself does not contain the definition of the file or the relation in a database that stores the data for the group. Rather, the DSD object discussed earlier contains this information. Therefore aspects of the user's interaction with the data (formatting of output and restrictions on input) are separated from details of how and where data is stored, which may differ in character between different types of data sources.

Several data groups can be combined within a form/report to support access to several data sources, possibly in different databases or in different kinds of databases and files. More specifically, groups in a form/report form a hierarchy that reflects the relationship between different data streams. Each group can have one or more children groups. In such a parent/child relationship, the data in the child group is related to and dependent on a record in the parent group; a field or set of fields in each child record must be equal to the corresponding field(s) in the parent record. In relational database terms, this simulates a join between the data in the parent and child groups. The application definer, simply by defining a parent/child relationship, achieves the following effects:

- When records are read for the child group, an implicit restriction is added to read only records for which the corresponding fields match.
- When records are inserted into the child group, RALLY automatically fills in the fields to match those in the parent record.
- When records are deleted in the parent group, RALLY can, at the option of the definer, delete all records in the child group, preserving the integrity of the database.

This ability to organize related data is crucial in the development of applications that use relational databases. Data normalization forces logically related data to be separated into multiple relations to avoid repetition or excessive functional dependencies within a single record. To display repeated data in its proper context and to display dependent descriptive data, a single form or report often relies on data from several relations. A typical example is an order entry form. Repeating data for the line items in the order are stored in a relation separate from the order header data. The order header data contains a reference to the customer, but descriptive information about the customer (such as name and address) must be looked up from a separate relation. Similarly, line items refer to products, but the descriptive information (product name and price) are looked up in yet another relation. A hierarchy of data groups in a RALLY form/report corresponds directly to the relationships among these relations.

At each level in the hierarchy of groups, the definer can allow or restrict insertion, deletion, and update of records. The definer can also define additional fields such as computed fields and aggregates. RALLY produces an instance of such fields for each record in the data group. For example, an aggregate field in a parent group will produce a set of subtotals, one for each record in that group.

Control break reports are also implemented in RALLY with data groups. The fields on whose values the control break is based are placed into a separate data group above the rest of the data. As with form/reports based on simulated joins, each level of control break can have aggregates and formatting attributes defined in the data group.

Nonrepeating fields, such as grand totals, are owned by a special group called the main group. The main group sits atop the hierarchy, owning the top-level data groups. This group can also be used for "pure forms" whose data is not automatically written out to a data source.

A special kind of data group, called a list-of-values group, offers a simple, nonprocedural method for ensuring referential integrity. From the point of view of the end user, the list of values assists in supplying a value for a particular field. The end user uses the RALLY command LIST_OF_VALUES (typically using a function key) to move the cursor from the field to the list. The user then moves the cursor to the desired value and uses the RALLY command SELECT VALUE to copy the value to the field. The application definer has the option to restrict the user to selecting a value that appears in the list of values. The implementation of a list of values is simple and consistent with the definition of other groups: a DSD describes the data that will appear in the list, and the data group describes the formatting of the data on the screen. Because the list-of-values data is independent of the other data on the form/report, the list-of-values group is neither a parent nor a child of the other data groups, but is realized as an independent sibling owned by the main group.

Escapes to Procedural Programming

RALLY tries to anticipate the features that will be required in applications and to provide the definer with the option to include those features. Fields, data groups, and form/reports as a whole are replete with options. But this is not enough. No collection of options will meet the requirements of all applications. Therefore, RALLY allows the application definer to escape from the nonprocedural confines we provide.

The VAX RALLY system offers the definer two levels of escape: an integrated procedural language, ADL, that runs within RALLY; and the ability to call traditional programs that run on the VAX system, independent of RALLY. Both ADL procedures and calls to external programs latch on to a RALLY application at various "hooks," called action sites.

Action Sites

A number of action sites are available at various levels in the application.

A simple example is a computed field that has an action site for a procedure which supplies the formula for the computation. In addition, action sites can be invoked before and after the user moves the cursor to each field or changes the value of each field; before and after insertions, deletions, and updates in each data group; before and after commits, rollbacks, queries, and invocation of the form/report as a whole; and at the explicit request of the end user. Action sites that occur before an event generally have the ability to prevent that event from taking place. For example, the before-deletion action site, under conditions specified by the definer, can forbid the user conditionally from deleting records in a particular data group. By using external programs or ADL procedures, the definer could call upon a system service to determine a user's login-identification, read records from an authorization file, and/or call a RALLY menu to allow the user to reconfirm, before proceeding with the deletion.

Because RALLY has direct control over the "action stack" that governs the flow control, action sites can be put to very powerful use. At any action site, the definer can call another RALLY action (for example, form/report, menu, ADL procedure, or external program), spawn a RALLY task, return to an existing task, "unwind" the action stack, or invoke a RALLY command (for example, COMMIT).

ADL Procedures

Although external programs can do things ADL procedures cannot, their effect on the RALLY application is limited to their ability to write their output parameters into RALLY fields or variables.

ADL provides a convenient way to define computed fields without having to link to an external program. Although there is some overlap between the abilities of external programs and ADL procedures, certain operations are better suited to ADL procedures. An ADL procedure can directly read and write fields and variables in the application; indicate that a validation has failed, preventing a data operation from going forth; invoke RALLY form/reports, menus, error messages, or help messages; unwind the RALLY execution stack to a specified point; and manipulate RALLY tasks. In addition, ADL can read, query, and write data through a set of built-in functions that closely parallel the operations permitted in form/reports. As with form/report groups, the specific definition of the location of the data is isolated in the DSD object.

Moreover, given the choice of using either an ADL procedure or an external program, a definer will find the ADL procedure significantly faster to implement. To incorporate an external program into a RALLY application, the definer must leave RALLY, edit the text of the program, compile and link the program, and return to RALLY. To use an ADL procedure, the definer can invoke the ADL editor and compiler from within the definition system menus, and test the results without leaving RALLY.

The syntax of ADL is a good example of how RALLY accommodates the definer's need for both simplicity and power.

ADL derives its syntax from Pascal in order to provide local variables, parameters with call by reference, conditionals, and loops. The syntax is relaxed for cases that do not use all these features:

- If an ADL procedure does not use parameters, the PROCEDURE statement may be omitted.
- If an ADL procedure does not use local variables, the BEGIN and END enclosing the procedure body may be omitted.

As a result, an ADL procedure that specifies the formula for computed fields — the most common use of ADL — can be written as a single Pascal statement. For example:

```
FORM_REPORT . COMPUTED_FIELD :=
     FORM_REPORT . INPUT_FIELD_1
     * FORM_REPORT . INPUT_FIELD_2;
```

Implementation of the Definition System

The definer of a RALLY application manipulates objects such as form/reports, menus, DSDs, external program links, and ADL procedures. Most of these manipulations are done in terms of data operations: creating, deleting, and modifying information in a "record" that represents each object.

The implementation of the definition system uses RALLY for its own forms, treating the application definer's objects as data. Thus the definition system is simply an example of an application built with RALLY, although probably much larger than the typical RALLY application. However, the definition system also includes tools, such as editors, and has an access method specifically designed for efficiently storing application objects in files. The code that supports this access method can also be called directly by the definition system code, or by ADL procedures in the definition system application.

The definition system uses its data about the application to assist the application definer. Whenever the application definer has the opportunity to connect one RALLY object to another (for example, at action sites), the definition system displays a list of values showing all the existing objects of the appropriate type. Whenever the application definer attempts to delete an object, the definition system checks for references from other objects; if it finds any such references, it warns the definer and displays a report that lists the referencing objects. Again, standard form/report features are used in connection with the specialized access method.

An interesting aspect of the way objects are handled as data is the way object names are handled. The definer regards the object's name as the primary key that uniquely identifies the object. However, to encourage the development of mnemonic names, RALLY allows the definer to rename objects. Therefore, RALLY internally identifies objects not by name but by an internal identifier not displayed to the definer. From RALLY's internal point of view, the name is just another attribute of each object that can be changed at will. From the definer's point of view, renaming an object automatically renames all references to that object.

The definition system uses a specialized form of escape to 3GL programs to support nonstandard form/reports. Called the 3GL access method, this technique allows the definition system to present information in tabular form even when the underlying data is not stored as a sequence of records. For example, the definition system uses the 3GL access method to display an Rdb/VMS record selection expression on a series of tabular form/reports: sorting, restrictions, and projections. Each "3GL DSD" is implemented by a single routine that can be called with one of several function codes. These functions correspond to the data operations that are supported in form/reports and in ADL procedures: get first record, get next record, insert, delete, update, commit, and rollback. Each such routine, in effect, implements its own access method, supplying data and effectuating data operations. The definition system can use the "access method" as any other data source, for example, to supply the data for a list of values.

The Run-time Environment

Mapping User Actions to Database Operations

The typical end user of a VAX RALLY application is not skilled in database concepts. Therefore, to aid this user, database operations should happen in a natural correspondence to the end user's actions.

The basic concepts that RALLY presents to an end user are very similar to those presented by the VAX TEAMDATA software, a data management tool specifically designed for exclusive use by unsophisticated end users. Specifically, the "data table" metaphor by which TEAMDATA operates is very similar to the mechanisms that RALLY uses. A data table evokes the classical form of a table to represent a relation. Rows in the table represent records; columns represent fields in the relation. The VAX RALLY run-time environment includes built-in commands with which the end user manipulates records or navigates between fields in a form/report. Function keys have been predefined for the commands most commonly used.

To delete a record, the user moves the cursor to a field in the record and invokes the DELETE RECORD command (typically by pressing the Remove key). To modify a record, the user moves the cursor to the desired field in the desired record and types the new value. However, the update is not communicated to the data source until the user moves the cursor off the record. Besides minimizing the cost of repeated updates, this allows the user to change several fields that are subject to cross-field validation imposed either in the database or by the RALLY application.

Transaction Management in Form/reports

RALLY uses Rdb/VMS transaction and locking mechanisms to extend TEAMDATA's straightforward data table metaphor from a single-user to a multiuser environment. Moreover, the designers wanted to allow many users to access the same data concurrently with minimal interference between users, and to do so with reasonably efficient performance.

The following brief review of the Rdb/VMS transaction and locking mechanisms will help in explaining RALLY's implementation of shared-write access in form/reports.

Rdb/VMS provides essentially two types of transactions: read-only and read/write.

A read-only transaction, as its name implies, permits only reading operations. It gives a "snapshot" of the state of the database as it was when the transaction started; later changes by other users are not seen in a read-only transaction. A read-only transaction does not take out any locks on the database and is affected only by those relation-level locks taken by other transactions with "exclusive" access.

A read/write transaction must be used to write to an Rdb/VMS database. In addition to various degrees of locking of relations, a read/write transaction locks individual records as it operates on them. If a read/write transaction reserves a relation for shared-write access, many transactions may read a given record, but only one transaction may write to a particular record. The mechanisms Rdb uses to ensure this are called "read locks" and "write locks." As a read/write transaction reads a record, it takes a read lock on that record. For so long as this transaction holds that lock, no other transaction is allowed to delete or modify that record. However, other read/write transactions may read the record, taking their own read locks on the same record. Read-only transactions are unaffected. When a read/write transaction writes to a record, a write lock is taken on that record. For so long as this transaction holds that lock, no other read/write transaction may read that record or write to it. Both read locks and write locks are held until the read/write transaction is terminated by a commit or rollback.

RALLY's simple, elegant "data table" model, if implemented simply by using a single read/write Rdb/VMS transaction, would thwart the goal of noninterfering, simultaneous, multiuser access. Recall that a read/write transaction takes a read lock as a side effect of reading each record. The mere displaying of a record in the table, even without the user attempting to modify it, would immediately interfere with other users' access to that record. Although several users could each read the record, their read locks would prevent any user from writing to the record. This would make the shared-write access virtually unusable. To decrease contention among users, RALLY implements shared-write access using two Rdb/VMS transactions: a read-only transaction for displaying records in a data table fashion, and a read/write transaction that is used sparingly as needed when the user performs data update operations.

Another difficulty is caused, however, by the use of the read-only transaction to display existing data for the user's perusal. Because the read-only transaction supplies a "snapshot" of the data as it was when the transaction started, it is possible for the displayed data to lag behind the actual state of the database. Other users may have written and committed changes to the record in the meantime.

To alleviate this problem of stale data display while avoiding the overhead of repeatedly reading from the database to check for updated records, RALLY employs a compromise. RALLY checks for and reports discrepancies only at the point where a user attempts to modify or delete a record. After RALLY warns the user that the record has undergone changes since it was read from the read-only transaction, RALLY redisplays the record with its current data. RALLY does this by reading the record from the read/write transaction. This read operation, called a select for update (SFU), is done as soon as the end user changes a single field in a record. This action by the user is the earliest indication RALLY has that a user's interest in a record is more than that of passive observation. Reading from the read/write transaction serves two purposes: it allows RALLY to alert the current user to any interim changes in the record, and by taking a read lock on the record, it prevents other users from making any further changes to the record. When the current user moves the cursor off the current record, RALLY writes the changes to the read/write transaction, taking a write lock on the record. Note that RALLY can still read and display the record for other users despite the write lock, because the other users are reading from their own read-only transactions. Finally, when the current user performs a commit, whether explicitly by using the COMMIT command, implicitly by means of a positive exit from a form/report, or as a result of the application definer's design, the read-write transaction is committed and all locks are released.

Advantages of Using RALLY for Its Own User Interface

The use of RALLY to implement the user interface for the RALLY definition system has resulted in several advantages. Despite some early bootstrapping difficulties, the use of RALLY within itself has noticeably improved the quality of the product. Any time we change the user interface for the definition system, we simultaneously exercise the definition system as well as the run-time system. The definition system has profited from the ease with which we have been able to incorporate into it the same features that are easy to develop in applications, notably, validation, lists of values of valid choices, and flexible flow control. As part of the ongoing development work on future versions of the VAX RALLY product, we have been able to experiment readily with the user interface for the definition system, for example:

- We have implemented prototypes of the menu structure of the definition system in which menus and forms have been changed to reflect better the relationship between the various attributes of each object. The time to implement these changes has been negligible, allowing the development group to spend appropriate amounts of time evaluating design alternatives, rather than on implementation details.
- We are studying a change in the way an application definer specifies the location of RALLY objects. In the current version, an application definer specifies for each object the start row, end row, start column, and end column. Under the proposal being studied, the definer would specify the start row and column, and the size in rows and columns. This change was easily prototyped without changing the way RALLY stores the information. We introduced computed fields for the size information and made

the end coordinates nondisplayed fields. Only two ADL procedures were needed, despite the fact that this change affected numerous forms.

- We are experimenting, by means of the 3GL access method, with form/reports that display data about the application in a nonstandard fashion. For example, we have designed a form/report that would list the location information about all the fields, groups, and text objects in a form/report. The 3GL routine to supply the data for this form/report adds spaces to the beginning of each object's name so that indentation reflects the depth in the hierarchy of form/report groups.
- We have prototyped a way to streamline the means by which an application definer works on related objects in a RALLY application. In the current version, the application definer works on one object at a time, returning to the menu tree each time to select a different object. The prototype took advantage of RALLY's "local function" feature, by which an application definer can give the user the ability to call a RALLY action at will by pressing a key. This feature would allow an application definer to press a key to edit an object named on the current screen. For example, if an application definer were editing a menu and were to move the cursor to the name of the form/report that is called as a choice from that menu, RALLY would suspend its editing of the menu and allow editing of the form/report.

The speed with which such changes can be made has allowed us to compress several cycles of design, implementation, testing, and reaction into the time ordinarily taken to complete a single cycle. The ability to respond substantively to user feedback is a major contribution to our efforts to improve VAX RALLY's user interface.

Also, our experience with the definition system, one of the largest applications ever built using RALLY, has given us valuable insight in evaluating RALLY and proposing new features. Lists of values is an example of a feature influenced by the use of RALLY by the definition system. Several features were added to lists of values for the benefit of the definition system to make the feature more useful for applications generally. These features include the ability to validate the user's typed input against a list of values, the ability for variables in the application to affect the set of records in the list of values, and the use of lists of values to translate keywords into code numbers.

Summary

To make application definers more productive, the VAX RALLY system is designed to be at once easy to use and powerful. VAX RALLY achieves these goals in several ways. First, it offers a small set of concepts that address those application features commonly needed by application definers. Second, RALLY gives the definer ways to combine small pieces and ways to move in and out of the nonprocedural environment of the definition system. Finally, the designers of this object-based system delineated objects based on knowledge of how the objects are likely to be used.

The VAX RALLY product's unified form/reports, comprising combinations of data groups connected to data sources, provide application definers the functionality most needed for interactive database applications.

The representation of an application by a set of connected objects allows programming to be treated as a data processing application that manipulates those objects. In particular, this representation has allowed RALLY to implement the user interface for the application definition system as a RALLY application.

The use of RALLY form/reports as the basis for the RALLY definition system has resulted in several significant advantages, both in anticipating the needs of users and in increasing our own productivity and flexibility in developing VAX RALLY.

General References

E. Horowitz, A. Kemper, and B. Narasimhan, "A Survey of Application Generators," *IEEE Software* (January 1985) : 40–53.

J. Martin, *Fourth-Generation Languages* (Englewood Cliffs: Prentice-Hall, 1985).

C. Date, *An Introduction to Database Systems*, Third Edition (Reading: Addison-Wesley, 1981).

VAX Rdb/VMS Guide to Data Manipulation (Maynard: Digital Equipment Corporation, Order No. AA-N036B-TE, 1985).

VAX RALLY Dialog User's Guide (Maynard: Digital Equipment Corporation, Order No. AA-GX89A-TE, 1986).

VAX RALLY ADL User's Guide (Maynard: Digital Equipment Corporation, Order No. AA-GX90A-TE, 1986).

Linda E. Benson
Michael Gianatassio, Jr.
Karen L. McKeen

VTX and VALU — Software Productivity Tools for Distributed Applications Development

Digital's VAX VTX product is a distributed information-retrieval tool that operates in conjunction with another tool, the VAX VTX Application Link Utilities, or VALU. These products enhance software productivity by providing components that work together to allow the development and integration of applications in distributed, heterogeneous environments. VTX and VALU provide the means for creating information services, providing network access to either centralized or distributed information, and building external applications through basic tools and programming interfaces. The development of distributed applications with VTX and VALU requires little or no knowledge of the underlying network.

The designs of VTX and VALU center on a distributed open architecture using the client/server model. This open architecture allows VTX and VALU applications to be integrated with others available through Digital's networking environment. The architecture enables these two products to provide a simplified development environment for applications. Within this environment, a developer can create distributed applications that allow geographically dispersed users to access information stored in geographically dispersed locations connected by a computer network. This flexibility allows the base services of a distributed information retrieval system to be extended into a more robust distributed system. In such a system, a developer can integrate applications with other software products or external computer systems.

The VTX Project Goals

The designers of the VTX product had to address a unique set of problems related to information access through a computer network. The central problem was how to efficiently distribute information on-line to a large group of people. The chief aspects of this problem were the following:

- There were neither means to access information stored in dispersed locations nor easy ways to alert potential users about it.
- Even if information could be reached, it was not well organized for ease of access.
- At the decentralized locations, information was usually maintained in different ways by those people most familiar with it.

These problems describe the information situation in most corporate business environments.

Therefore, the challenge for the VTX designers was to determine how a corporation handles information flowing between different locations. Note that information in this case could be anything from policies and procedures manuals, to job postings and travel schedules, to CAD/CAM drawings and technical documentation. A major goal of the designers was that a minimum of special learning should be required by people accessing the information; browsing through it should be as simple as using a telephone.

This goal caused the VTX architects to examine various systems with these characteristics, including public videotex systems, that addressed the problems listed earlier. A main feature of public videotex was that basic navigation through the on-line information system was simple for users. They could easily locate information and then rely on the system to quickly and easily access that information. Public videotex systems were also distributed systems: users were geographically dispersed, and the information accessed was located in multiple information "stores" in a computer network.

Since public videotex was a well-accepted system that was also easy to use, the architects chose it as the basic model on which to build the VTX product.

Figure 1 presents the architects' view of a simplified model of information flow within a corporation. This model would be refined as project goals were clarified.

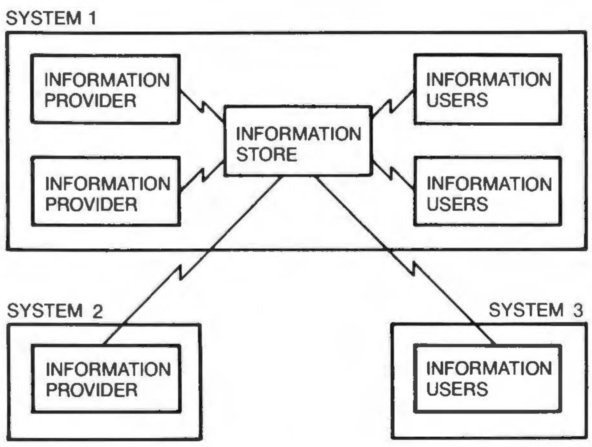


Figure 1 Information Flow Model

Although ease-of-use was a primary goal for an information retrieval system, other major goals included the following:

- Information access must be fast.
- The product must accommodate access from a variety of desktop systems and terminals from different manufacturers.
- The product should be protocol neutral so that information, regardless of its format or presentation-level content, could be stored in a single data store.
- The product must support a distributed environment in which the information, as well as its users and information providers, might all be geographically dispersed.

Guided by these goals, the designers built a prototype that was then tested by users within Digital to determine its ease-of-use factors, performance, and general acceptability. Based on its success as a prototype, the development of the product began, following the goals described above. Somewhat later in the development phase, the team added goals aimed at making the product extensible, thus taking advantage of the open architecture. Extensible in this case simply meant the ability to enlarge the tool-kit nature of the product so that an application developer could expand the system by interfacing with other products and environments. As a result, designers identified some growth areas for these products that took advantage of their flexible open architecture.

These additional goals led to the concept for the VALU product. Although VTX would provide the base services for a distributed information retrieval system, those services had to be expanded to interact with other applications. VALU was conceived as a tool kit that would allow application developers to enhance applications built with the basic system provided by VTX.

- Enhanced tools and environments for the information providers
- Tools for acquiring and incorporating information into the VTX-based system
- Integration with non-VTX applications through interfaces that require no knowledge of the underlying network by the application developer

Building the Base VTX System

This section discusses the characteristics of a distributed architecture, how it facilitates application development and integration, and the methods for building upon it.

Characteristics of a Distributed Architecture

Distributed connotes dispersion, spreading out and placing things in different places. A distributed application comprises two or more application components, separated from each other, but working together to form the application. An application component is a self-contained program that executes independently of other application components.

Application components may reside on different CPUs or on the same CPU. In either case, these components need some means to communicate with each other. The communication means chosen may vary depending on the locality of other application components. For example, components residing on the same CPU may communicate through shared memory, whereas those on different CPUs may communicate over a network. The design of an application component is independent of the locality of other components. At run time, the software supporting the system can sense any difference in localities and choose the appropriate communication means.

Since the locations of application components are transparent to the design of the application, these components may be distributed across heterogeneous, or mixed-vendor, environments. For example, DECnet software extends its connectivity to heterogeneous environments through SNA and X.25 networks. Therefore, the components of a VTX/VALU system may also be distributed across these environments.

Client/Server Relationship

The components of a distributed application have a client/server relationship. As consumers of resources, clients initiate requests to servers; as providers of resources, servers respond to requests from clients.

Clients and servers generally interact according to a request/response protocol. Since requests and responses may be formulated over multiple messages, a "token" is used to regulate whose turn it is to communicate. The application component that possesses the token has the right to communicate. The completion of communication is signaled by the passing of the token, either explicitly by flags within the message or implicitly by the message type.

Servers may communicate with other servers. The server that initiates the communication then becomes a client to the other server. If a server communicates with another server on behalf of its client, that server is called a broker. Figure 2 illustrates this broker relationship. Brokering facilitates application integration and allows clients transparent access to any application available throughout the network. How the VAX VALU product integrates applications through brokering services will be discussed later.

Regardless of which communication means is used for application components, a set of rules in the form of protocol messages must be defined to specify how functions are distributed. These rules are called the application protocol.

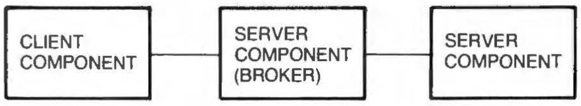


Figure 2 Client/Server/Broker Relationships

Application protocols define how functions that are specific to the application are distributed across its components and the rules for component interaction. Although the application protocol is independent of the communication means, the protocol may require certain characteristics, for example, full-duplex communications.

Application protocols must be invisible to the application developer. In the VTX and VALU products, callable application protocol libraries are implemented to increase the productivity of the application developer by

- Creating a single library that supports the application protocol and is shared by all application components requiring support for the protocol
- Having the application developer learn a simple, higher level call interface rather than all the details of the application protocol
- Defining a clear interface for integrating applications into a distributed environment
- Insulating the application from changes in the application protocol
- Resolving incompatibilities between application protocol versions in application components
- Insulating the application developer from the communication means used between application components
- Facilitating the development of applications that can be accessed by simultaneous users

How the Architecture Achieves the Project Goals

Fast information access in a distributed environment is achieved by making VTX available over networks with the DECnet architecture. This architecture extends connectivity to multivendor environments, thus achieving the goal of accessing dispersed information. For example, the DECnet/SNA Gateway and the packet-switching interface products (X.25) provide access to a great variety of non-Digital environments.

The architecture allows transparent access to heterogeneous environments through the brokering capability of servers. Users can navigate transparently to other VTX servers or to applications that have been integrated into the VTX environment. Applications integrated into that environment can be developed independently of whatever input devices the users have. Therefore, application developers can make their applications available immediately to any user on the network having the standard VTX client. No additional software needs to be installed on the client systems.

The features mentioned above provide full support for processing and storing data in a truly distributed, heterogeneous fashion. By allowing transparent navigation to servers and applications, the architecture can retrieve data stored in any format from any point throughout the network.

Components of the VTX/VALU Product Set

The VTX and VALU product set includes a collection of application components interacting over the network. Their foundation is based on the distributed architecture illustrated in Figure 3.

The store of information referred to earlier is contained in the VTX information base, a hierarchical system of pages called an infobase. The infobase contains presentation information that users can navigate through using menus and keywords. On-line updates are allowed because the infobase is shared among the infobase, update, and VISTA servers.

The terminal control program (TCP) is responsible for presentation management and parsing users' requests according to the specific input devices being used. The TCP maps those requests to specific VTX function requests, which are then sent to the information server. The TCP and the information server communicate through the DECnet software using an application protocol called the videotex access protocol (VAP).

Infobase servers communicate with other infobase servers or with applications on behalf of the TCPs. This communication is transparent to the TCPs, thereby providing transparent access to them. All communication between infobase servers and applications is through the VAP application protocol.

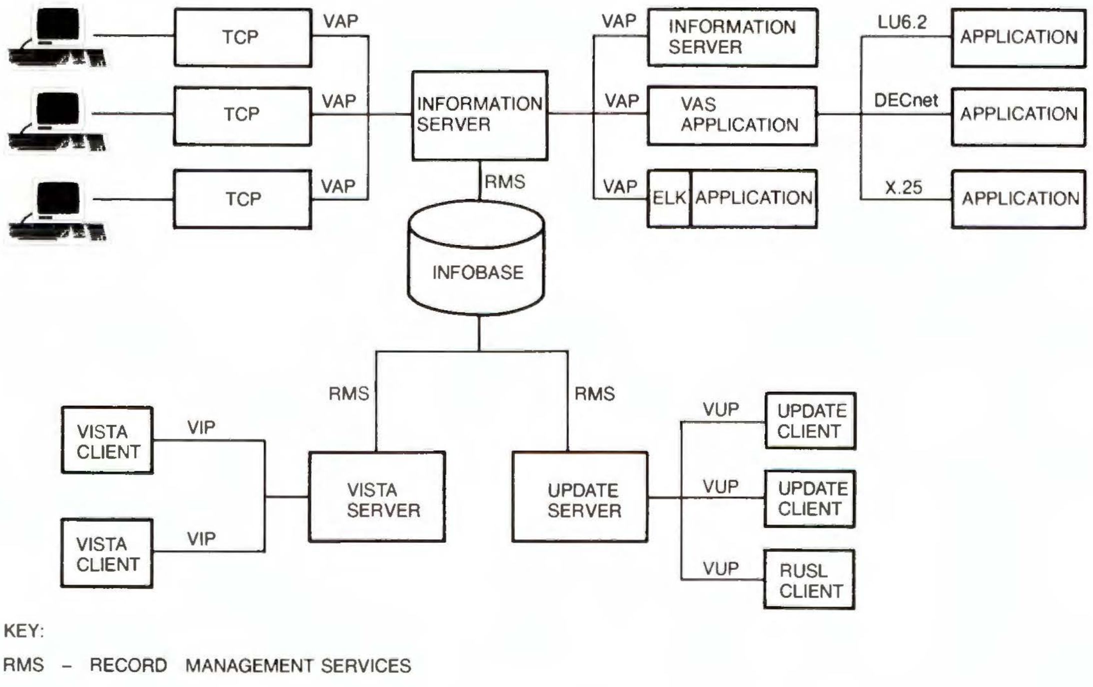


Figure 3 VTX and VALU Product Architecture

The external link kit (ELK) interface is an application protocol library used by programmers to integrate applications into the VTX environment through the VAP protocol.

The VTX application service (VAS) is a distributed application integration tool. VAS allows access to applications on other computer systems through both DECnet and non-Digital networks. The productivity gains of developing and integrating applications into the distributed environment of VTX using VAS is discussed later.

The VTX infobase structure tool and assistor (VISTA) is a distributed application development tool that displays a graphical view of an infobase to assist a user in creating and managing an infobase. VISTA clients are information providers who interact with a VISTA server to perform those tasks. All communication between VISTA clients and servers is by means of the DECnet network using an application protocol called the videotex information provider (VIP). Using VISTA yields productivity gains that are discussed in the next section.

Update clients are information providers who interact with an update server to create and maintain an infobase. The update server uses a command-oriented interface. All communication between update clients and servers is through the DECnet network by means of an application protocol called the videotex update protocol (VUP).

Application developers can also use RUSL (remote update server link), an application protocol library used to create infobase management tools as well as to allow the updating or populating of an infobase from a program. The RUSL application protocol library was built to support the VUP protocol. The VTX update client was built using this protocol library.

Tools Provided for Application Development

This section discusses two tools provided by the VTX and VALU product set to enhance the productivity of application developers building distributed information systems. VTX information systems can be grouped into two classes: those simply providing information to users, and those interfacing with other applications to enhance and expand on the information itself. The first tool, VISTA, addresses the needs of application developers building the first type of information system. The second tool, VAS, provides the necessary capabilities that allow application developers to easily build interfaces to other applications.

VTX as an Application Development Tool

The VAX VTX base product provides the tools for quickly and easily building a distributed information application. The open architecture allows an application developer to easily extend the application by adding new applications and support for heterogeneous environments as the requirements change. These extensions are discussed later in the section Application Integration Using VAS.

The basic VTX components allow an information provider to create an infobase. The infobase is the application; the information provider is the application developer. Policies and procedures manuals, sales and competitive information articles, reference manuals, jobs books, training schedules and course descriptions, CAD/CAM drawings, and newswire stories are examples of information that can be organized, maintained, and delivered as VTX information applications.

With the simple information-based application, VTX relieves an information provider from having to know specific details of the underlying communications and the terminals. Thus, the information providers can direct their attention to the content of the information and can more easily design the structure by which an information user accesses the infobase.

VISTA

VISTA is a tool to increase the productivity of the information provider when creating a distributed information system using VTX. VISTA helps a naive information provider to become productive very quickly and allows an experienced information provider to remain productive. VISTA provides a simple graphical interface for the naive user, yet also has a command-line interface for the more experienced user. The productivity of creating applications increases because VISTA is easy to use.

VISTA uses the client/server model to allow one VISTA server to maintain the actual VTX infobase files. One or more information providers can access that VISTA server through the VISTA user-interface program (VISTA client). Building on the foundation of a distributed architecture allows information providers to be dispersed geographically. VISTA can coordinate multiple information providers working on an infobase by allowing them to reserve portions of it for updating.

VISTA uses a simple graphical interface that allows an information provider to quickly design the layout of a VTX infobase. The picture displayed represents the hierarchical nature of the infobase, much like an information provider would imagine the infobase menu structure to be. VISTA improves and enhances the infobase development process by allowing the information provider to design the menu structure right on the terminal instead of constructing it first on paper. Figure 4 shows a sample of a VISTA screen with a menu structure.

An information provider using VISTA builds a VTX infobase by selecting options from the strip menu appearing at the bottom of the screen. The large work area above the options menu displays the current state of the infobase as the information provider continues to work. The single box at the top of the work area depicts a current menu. Any number of boxes drawn just below the current menu show the menu choices from the main menu. Any of those menu choices may themselves be menus. The menu structure of the VTX infobase can be modified through simple add, delete, and cut-and-paste operations.

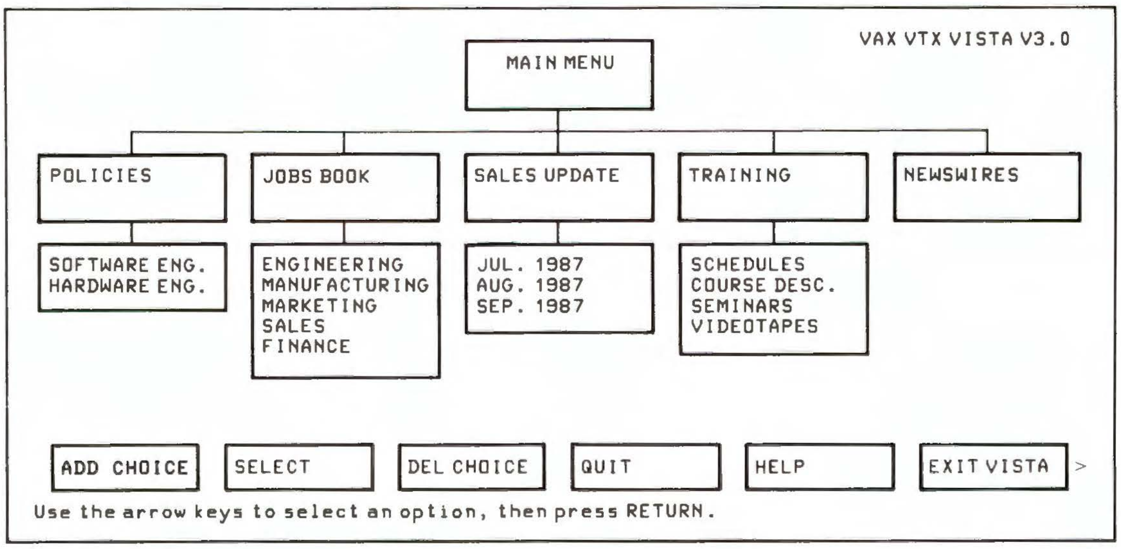
Each box at the top and across the center of the screen represents a particular page in the infobase. In an application development environment, the VISTA options allow an information provider to easily specify all the necessary information for each page. The information provider can invoke an editor to supply the text for any page without leaving the environment. VISTA builds the text of menus according to a default style and allows that style to be modified for any menu page. Its simple forms interface allows the information provider to specify additional information for any page. This page information is grouped into categories of similar items, each with its own form. Figure 5 shows a VISTA form.

Once the information provider has created the menu structure, VISTA handles the process of building that infobase from the picture. Each page in the menu structure is converted from graphical format to infobase format through the VISTA server. VISTA handles the underlying complexities of page generation, such as page numbering and the association between a menu page and its choice pages.

Once the information provider has initially developed an information application using VISTA, the tool can continue to be used to enhance, extend, and maintain the application.

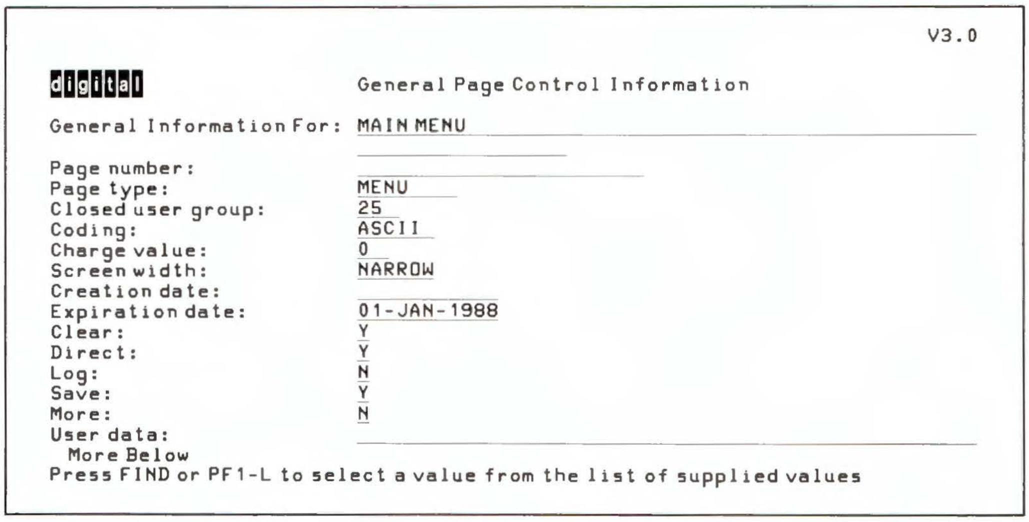
Extending the VTX Application

Using the VAX VALU product, the information provider can extend the basic application into a more interactive one by using the VALU tools for two-way access to the infobase. Not only can the



Use the arrow keys to select an option, then press RETURN.

Figure 4 Sample VISTA Menu Structure



```
                                                                V3.0
digital                      General Page Control Information

General Information For:     MAIN MENU

Page number:
Page type:                   MENU
Closed user group:           25
Coding:                      ASCII
Charge value:                0
Screen width:                NARROW
Creation date:
Expiration date:             01-JAN-1988
Clear:                       Y
Direct:                      Y
Log:                         N
Save:                        Y
More:                        N
User data:
  More Below
Press FIND or PF1-L to select a value from the list of supplied values
```

Figure 5 Sample VISTA Form

application distribute information to users, it can also collect information from them, process it, and return the results. Using VALU allows an application developer to define the flow of control for infobase access by a user. VALU also allows simple connections to external applications that can provide information to the system. Some examples of external programs are on-line ordering and registration systems.

The next section describes the VAS component of VALU. VAS is a powerful application development and integration tool that provides the functionality to connect the VTX system with applications on DECnet, SNA (LU6.2), and X.25 networks. VAS was built on ELK, the application protocol library that is provided with VALU.

Application Integration Using VAS

VAS is a flow-control and integration layer between the VTX/VALU environment and external applications. Figure 6 illustrates how applications are integrated using VAS.

VAS simplifies application development and integration by providing a fourth-generation language for using VAP. The VAS language is specialized to provide the functions of VAP and to facilitate the integration of external applications. VAS applications are organized into scripts called transaction definitions. Application flow control can occur by transaction definitions transferring processing to other transaction definitions.

The VAS language consists of eight verbs used to

- Display information from the infobase of the server (optionally merging data from VAS), collect user responses, and make flow-control decisions based on user requests
- Declare local and global variables
- Manipulate the contents of variables
- Pass variables and state information to external applications (or to local user-written routines that have been dynamically loaded in the VAS image) and receive responses
- Make flow-control decisions based on responses from user input, external applications, and user-written routines
- Log the contents of variables

Interaction with External Applications

VAS interacts with external applications over communication channels using its own request/response protocol. A request contains current state information about a user, for example, which transaction the request was made from, the contents of variables, what operation is being requested, and time-stamp information. A response contains updated variables from the processing of the application.

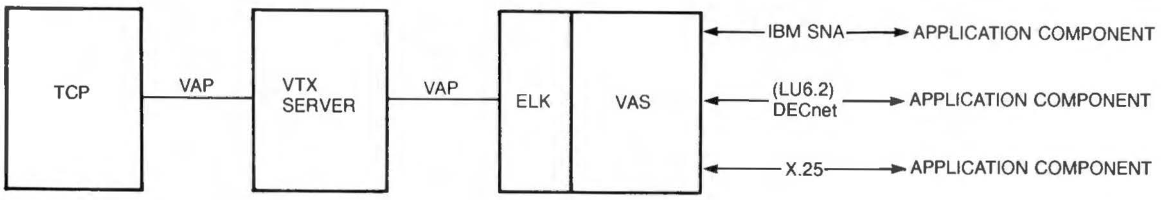


Figure 6 Application Integration through VAS

VAS transaction definitions specify the names of the communication channels over which requests to applications are made. A single VAS transaction may interact with multiple applications over various communication channels. The VAS operator dynamically associates communication channel names to specific communication types and specific applications. VAS has built-in support for communicating with applications over the DECnet, SNA (LU6.2), and X.25 networks. The same request/response protocol is used over all communication types. Transaction definitions are written independently of the communication channels used; the application developer requires no knowledge about the network.

Communication channels are shared by users and may have multiple outstanding requests; however, each user can have only one outstanding request. VAS manages the sending and receiving of all requests. These activities include suspending the execution of a transaction definition, timing requests, receiving a request and identifying which user's transaction definition to resume, and extracting the contents of a request into local variables.

Using the concepts of the request/response protocol over communication channels, a VAS application developer can build an application that uses a consistent programming interface to communicate with a variety of heterogeneous environments. Let us examine some of the details that VAS handles for the application developer.

VAS and SNA (LU6.2)

VAS uses the DECnet/SNA Gateway and the VMS APPC/LU6.2 products to communicate with the SNA environment, as illustrated in Figure 7.

The application developer using VAS requires no knowledge of the DECnet/SNA Gateway and VMS APPC/LU6.2 products, or of the IBM environment. The CICS transactions on the IBM system need to conform only to the request/response protocol.

VAS and the X.25 Environment

VAS can communicate with any packet-mode data terminal equipment (DTE) that is connected to a packet-switching data network (PSDN) by using the VAX PSI product. The PSDN provides task-to-task communication between any two computers connected to an X.25 network. Therefore, the environment is heterogeneous in nature. The VAS developer needs no knowledge of the PSDN, the VAX PSI product, or the remote DTE being accessed. Applications written on the remote DTE need to conform only to the request/response protocol.

VAS and DECnet Applications

VAS can communicate with other DECnet applications using either transparent or nontransparent task-to-task communication. Once again, the VAS application developer needs no knowledge of the DECnet software. External applications written on the remote system need to conform only to the request/response protocol.

Handling Simultaneous Users

The VAS developer writes transaction definitions as if they were synchronous and for a single user. After VAS compiles and loads the transactions, they become available for simultaneous users. VAS interleaves user activity by suspending users whose transactions are currently performing asynchronous activities, for example, waiting for the TCP to respond to the last page displayed, or

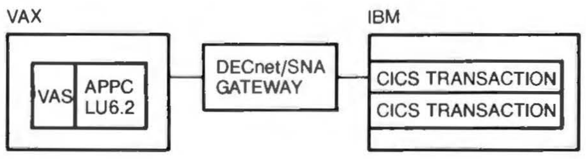


Figure 7 VAS Integration Using SNA (LU6.2)

waiting for an application to respond to a request. In other words, while one user is waiting for some sort of I/O to complete, VAS processes another user's request.

Since VAS automatically handles simultaneous users, simultaneous requests may be generated to the same application. However, developing applications to service simultaneous users can be complex. Therefore, VAS has built-in features that allow single-user, synchronous applications to service simultaneous users. In that way the burden of developing a simultaneous-user application is removed from the application developer. These activities are all accomplished through the subchannel feature of the communication channel. The VAS operator can start multiple copies of the same application on a single communication channel. When requests are made over the channel, VAS allocates a copy of the application that is not currently being used. If all subchannels are busy, VAS holds the request until a subchannel becomes available. The subchannel feature creates a pool of identical applications which can be distributed across all communication types. From the VAS transaction definition, this pool of applications acts like a single communication channel. Figure 8 illustrates these subchannel capabilities.

Creating High Availability Computing Environments

VAS functions are managed without interrupting any active users. Such functions include starting and stopping communication channels, opening and closing log files, loading new transaction definitions, modifying the contents of global variables, and changing the association between channel names and communication types.

The VAS application developer can make updates to transaction definitions and load them dynamically. Users accessing the transactions before they were modified will continue to use the older version of the transaction definitions. New users who connect to VAS will immediately start using the updated transaction definitions. When there are no active users on the older transaction definitions, they are automatically unloaded.

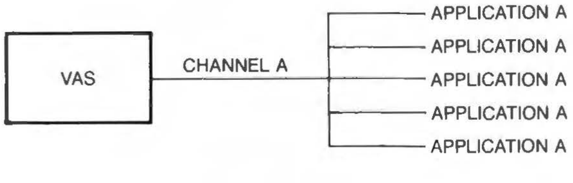


Figure 8 VAS Subchannels

Sample VAS Application

This section contains a sample VAS application that displays a form page to the user and then sends a data block (using the REQUEST step) to pass two fields with initial values (specified by the DFLD variables) to a CICS transaction for processing. Upon returning from the CICS transaction, the VAS application directs that page 102 be displayed from the VTX infobase to the user. This sample VAS application could be part of an interactive banking application that gathers data from the user and then validates it (for example, a user's bank account number and access code) before allowing access to the system.

```
TRANSACTIONfirst_trans/ENTRY
    ='accesscode'
   BEGIN
     DISPLAY '1025'
     REQUEST checkcode sna_chan1
       BEGIN
       RFLD-1 DFLD1/LENGTH=6
       RFLD-2 DFLD2/LENGTH=15
     END
   EXIT/PAGE='102'
END
```

The channel to the SNA gateway has been established with a command of the following format:

```
VAS>START CHANNEL sna_chan1/SNA=
    (GWY=sna_gwy,
    ACC=cics_access,
    TPN=cics_trans1)
```

GWY is the name of a DECnet/SNA Gateway, ACC is the access name on that gateway that allows access to CICS, and TPN is the name of the IBM host transaction to be invoked.

Sample Distributed Application Using VTX/VALU

The following example illustrates an application that was built using the VAX VTX and VAX VALU products to handle document publishing with an integrated free-text search product. This example highlights some capabilities of these distributed products and illustrates the benefits of building applications utilizing the distributed base system.

The primary functions of this application provide

- A periodical called Sales Update on-line for corporate-wide distribution
- The capability to supply users with a free-text search feature using a third-party application called BASIS
- The capability to supply users with a hardcopy formatted version of any article in the periodical through an integrated mail interface

In this application the screens of information are formatted by a preprocessing application that creates the VTX infobase. This same application also provides data to the BASIS database and supplies files formatted for hardcopy output.

The information user is presented with a menu of categories of Sales Update articles, along with an option to search through them for a particular text string. If the user selects a search option, the VALU application will pass the string to the BASIS application, which returns to VALU one or more article IDs that match the search criteria. VALU then creates a menu dynamically with associated title strings to help identify the articles located. Subsequently, the user chooses a menu selection and the resultant article is displayed.

A further extension to this application gives the user the option of mailing the current article being viewed. Figure 9 gives a view of the components of this distributed document publishing system.

This example demonstrates some of the most important productivity benefits to an application developer.

- A single application program utilizing VTX and VALU can be accessed by a network of users without regard to the asynchronous environment and the need to support a large number of users.
- An information retrieval-only application can easily be extended to interface with external products without changing the user interface or disrupting the information retrieval "service."
- The applications provide VTX-like access that is consistent with the information retrieval access to provide extended capabilities to the users. This extension allows the application to free the users of the system from having to learn a new interface to the newly integrated product. In fact, the integration to another product may be virtually transparent to users.
- The application can support both softcopy (on-line) and hardcopy distribution from a single source file.

Summary

Working together, the VAX VTX and VAX VALU products provide a rich set of software produc-

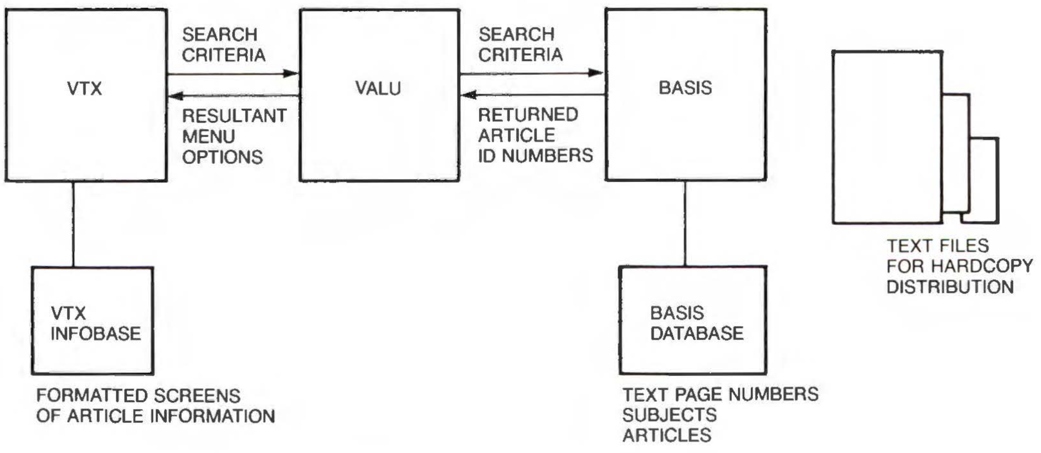


Figure 9 Distributed Document Publishing Application

tivity tools and programming interfaces. These products enable an application developer to construct both centralized and distributed applications used in both homogeneous and heterogeneous environments. Because of the range and flexibility of these tools, the resulting systems can differ significantly according to the function sets utilized and configurations selected. For example, with these products, a simple information retrieval or transaction-based system can first be built and then evolve into a more complex system based on the concepts of the VTX/VALU open distributed architecture. The ability to expand a system is essential; with the capabilities provided through VTX and VALU, this evolutionary system model is easily achieved.

General References

VAX VTX Documentation Kit (Maynard: Digital Equipment Corporation, Order No. QL031-GZ-V3.0, 1987).

VAX VALU Documentation Kit (Digital Equipment Corporation, Order No. QL035-GZ-V2.0, 1986).

J. Morency, D. Porter, R. Pitkin, D. Oran, "The DECnet/SNA Gateway Product — A Case Study in Cross Vendor Networking," *Digital Technical Journal* (September 1986): 35–53.

P. Beck, J. Krycka, "The DECnet-VAX Product — An Integrated Approach to Networking," *Digital Technical Journal* (September 1986): 88–99.

Ronald F. Brender
Bevin R. Brett
Charles Z. Mitchell

Pragmatics in the Development of VAX Ada

The software tools and techniques (pragmatics) used daily by the VAX Ada developers significantly contributed to increases in product performance and developer productivity. Approximately 500,000 lines of code were written for this project. Of particular interest in this project's development is the automation of the coding process, instrumentation of the compiler, built-in consistency checking within the compiler (self-checking), and the use of self-describing data structures. This paper gives examples of how these tools and techniques were used in the development of the compiler. However, these tools and techniques can be applied to a wide range of software development efforts.

Software engineering literature to a great degree focuses on design and implementation methodologies, as well as on tools to go with them. Little is said, however, about day-to-day tools and techniques that can also significantly impact the productivity and effectiveness of a development team.

The development of VAX Ada involved writing approximately 500,000 lines of BLISS source code. This paper discusses some of the tools and techniques that have been important over the course of that development. The tools and techniques fall roughly into the following categories:

- Automation
- Instrumentation
- Self-checking
- Self-description

We would like to suggest that many of these tools and techniques could be useful to any software developer and could be applied to any project of significant size.

Automation

During the development of VAX Ada, we wrote support code to automate various aspects of the coding process and to help with day-to-day development activities. Our interest was never in tools or automation techniques for their own sake. Instead, we were interested in tools and techniques that would minimize the time spent on routine or duplicate activities and would maximize the amount of time available for interesting technical problems. Thus, we balanced the value of each tool and automation technique against the time it would take to build the tool or develop the technique and use it. We wanted to spend most of our time developing VAX Ada.

These are some examples of activities we automated:

- Production of error-message information held in common between the compiler and the user documentation
- The task of creating and entering tests into the VAX Ada test system
- Compiler builds and check-in procedures
- The process of managing multiple versions of the compiler
- Some debugging tasks
- Key algorithms within the compiler

The first two examples are described in more detail in the following sections. The last example is described at the end of this paper in the section Self-description.

Production of Error-Message Information

VAX Ada has close to 1,000 distinct error messages. One of the VAX Ada user manuals lists all the error messages in an appendix. The usual method for producing error messages and documenting them is to create two source files: one source file to be maintained by the compiler developers and processed by the VMS message compiler, and one source file to be maintained by the writer and processed by the documentation processor.

This method quickly became difficult for us to manage. Messages were continually added as the VAX Ada compiler was being developed — even during the final stages of the development cycle. Because documentation is written and reviewed concurrently with compiler development, the writer and editor of the user manual needed to have a matching set of messages in the documentation for draft reviews and final production. The writer and editor also needed to be able to suggest wording modifications as messages were written, including wording modifications to the messages added late in the development cycle.

To keep the two sets of sources synchronized, we chose to automate. We wrote a processor that accepts a superset of the language accepted by the VMS message compiler as input. The additional language constructs allowed us to write one source file containing all the messages and any descriptive text appropriate for the documentation. Both the developers and the writer were allowed access to the file. We then used the processor to produce two output files: one file containing all the messages and the coding required for the message processor; and one file containing all the messages, any descriptive text, and the formatting constructs needed for the documentation processor.

Our processor saved us from having to review two different sources at already busy times in the development cycle. This approach also allowed the user manual appendix to be generated immediately for each new version of the compiler.

Creating and Entering Test-System Tests

An important task too often neglected is the preservation of tests written during development for later use by developers and maintainers. We observed during the development of VAX Ada that the number of tests added to our test system varied inversely with the difficulty of adding them. Our objective in automating the tasks associated with adding tests to our test system was to make sure that these complex, and important, tasks would be done routinely and accurately.

For example, we developed

- Command procedures to help create a test that followed project conventions
- Command procedures to automatically insert a test in the test system
- The ability to mark comments within a test as keywords, and then automatically read the keywords comments and enter them as attributes of the test in the test system
- Support for all major classes of tests (compiler, project library manager, debugging support, etc.), so that no major areas of testing were neglected

Instrumentation

The richness of the Ada language presents a number of challenges to compiler writers. One of these challenges is to achieve good compiler performance. We found that instrumenting the compiler to measure its use of resources was an important factor in developing a high-performance product.

We used general-purpose tools such as the VMS Debugger and the VAX Performance and Coverage Analyzer extensively during the development of VAX Ada. These tools were also very useful in improving performance. However, our specialized instrumentation helped us analyze the behavior of the compiler at a level relevant to the development strategies we were using; we could then better understand how these strategies were working. Many performance problems would never have been identified had it not been for our specialized instrumentation. As a result of our positive experiences, virtually every component in the compiler that manages a resource is instrumented to gather detailed performance statistics.

The following sections show how instrumentation is used to

- Provide data for design decisions affecting compiler performance
- Regulate the behavior of the compiler
- Provide information on the behavior of the compiler during maintenance and debugging

Although some of this instrumentation code is present in the production version of the compiler that is shipped to customers, most of it is conditionally compiled into only the debugging version of the compiler.

Instrumentation as a Performance Design Aid

As the Ada language itself was being developed, we began to research the novel aspects of implementing an Ada compiler and developed a breadboard compiler as a vehicle for our research. Because the breadboard compiler had been instrumented extensively, we used it to collect data to guide the design of the eventual product. A number of design decisions were made as a result of the data collected during the research period. The role of instrumentation with respect to the compiler's use of virtual memory is examined in particular in this section.

One major resource problem in the breadboard compiler was the vast amount of virtual memory required to compile some representative Ada programs. The amount was often an order of magnitude more than was acceptable to meet our compiler performance goals. Our instrumentation data revealed that the tree structure used to internally represent the Ada code occupied most of the memory. Therefore, a finer analysis of the data was performed based on frequency of occurrence and size of the individual kinds of tree nodes.

For example, we instrumented the tree node creation routine to count the number of nodes of each kind that were created. The counts and the number of bytes occupied by each node of a particular kind were displayed in the compilation listing file.

An analysis of the data showed that relatively few kinds of nodes occupied most of the space and suggested a number of improvements in the design of the tree to reduce the frequency of such nodes. The combined effect of these improvements halved the memory required for representing the tree for typical Ada units.

Further instrumentation showed that the addition of code-generation information to the tree representation substantially increased the tree size. These measurements suggested that memory usage could be decreased by recycling memory as soon as possible. An "inside-out" code-generation scheme was devised for our version 1.0 compiler. With this approach, object code is generated for the most deeply nested subprograms in a compilation unit first. The entire tree representation and code-generation information for a subprogram is no longer needed once the code has been generated, and can be freed before the code is generated for the next subprogram. Thus, the memory is available for reuse by the next subprogram. This approach reduced the amount of memory required to compile a typical Ada unit by a factor of two or more. This improvement, combined with the tree modifications mentioned previously, made it possible for us to meet our compiler performance goals with respect to virtual memory usage.

Instrumentation to Regulate Compiler Behavior

We also used instrumentation data gathered during a compilation to actually modify the overall flow of the compiler, and thus improve the compiler's performance. In particular, the compiler uses instrumentation data to modify its behavior according to the availability of memory. This kind of optimization is often seen in computer operating systems and in general manufacturing processes, but rarely seen in software tools such as compilers. This section describes the use of instrumentation data to diagnose and solve a paging-rate problem we detected during the development of the compiler.

The VAX Ada compiler consists of a number of phases that process the internal tree representation of Ada source code in a series of tree traversals, or walks. Walks in the semantics phase modify the tree representation to reflect the semantic meaning of the Ada code. Later walks, prior to optimization and code generation, add code-generation information to the tree.

Each of these walks is instrumented to show the amount of CPU time, elapsed time, page faults, and I/O operations involved. An analysis of this information during the development of VAX Ada showed that a very large number of page faults often occurred for typical program units. Even with larger than normal working sets, the paging rate was high enough to significantly increase the load on the system, thus affecting overall system performance and responsiveness for all users. Comparison of the paging rates with the same data for other parts of the compiler, and against the totals for the whole compilation, showed that a very large proportion of the page faults occurred during the walks that added code-generation information.

The trouble with any "static" solution to this problem is that page faults are a property of the amount of physical memory available to the compiler. The amount of physical memory varies based on both the VAX hardware configuration and the use of that hardware by other VMS processes running concurrently with the process executing the compiler.

In an effort to solve this problem, we measured the size of the tree for typical Ada subprograms. We found the tree size to be significantly smaller than the size of the code doing the individual tree walks. Furthermore, the code for the tree walks was larger than typical VMS working sets. Thus, the code for each walk was paged out by the subsequent walk and then paged back in again for the next subprogram. We concluded that the high paging rates were caused by our inside-out code-generation approach, which was designed to minimize the use of virtual memory.

To reduce the paging of the code, we chained together the trees for sets of subprograms and did each walk across all the elements of the set before applying the subsequent walk to any of them. This approach is contrary to the earlier goal of reducing memory usage by doing one subprogram completely before doing the next one. However, in this context the earlier goal is more accurately stated as "keeping the memory usage to within the amount of memory that is available."

As a result of our observations, we also made the compiler "self-correcting" in a release following version 1.0. We instrumented the compiler to measure the amount of virtual memory available, the amount of physical memory available, and the pre-code-generation size of each subprogram's trees. In addition, very conservative heuristics estimate the additional memory required for the code-generation information for each subprogram. Together, the measurements and heuristics are used by the compiler to build the largest possible set of subprograms that do not present a danger of exceeding the available virtual memory. Furthermore, the sets are chosen so that the code for the largest phase plus the size of all the trees for the subprograms in the set are less than the size of the working set extent of the VMS process.

This modification successfully lowered the paging rates of the compiler, hence improving elapsed time and system performance. The exact numbers vary according to the actual VAX hardware configuration and Ada code being compiled. However, figures for the code-generation phases were often halved, resulting in 30 percent or more overall improvement for the whole compilation.

This dynamic measurement of working set, virtual memory, and tree size and the subsequent tuning of the selection of sets to the process's available resources means that all resources — large or small — were fully exploited. This technique is applicable for enhancing the performance of any compute-bound programs that also use significant amounts of virtual memory.

Instrumentation as a Debugging and Maintenance Aid

In addition to using instrumentation to obtain resource measurements, we have used it to debug the compiler. We have also found it to be a useful maintenance aid.

Instrumentation data is read by calling one of a number of routines either from the VMS Debugger or from code triggered by an event. (Events are special places in the compiler code.) The routine displays the instrumentation data on the terminal (so the programmer can see it right away) and in the listing file (for post-mortem examination). The debugger or event-driven routines are capable of producing human-readable listings of large and complex data structures. The listings help simplify the task of debugging the compiler, as it can be very time-consuming to examine directly a very complex data structure, such as a tree, with a general-purpose tool like the VMS Debugger. (An example listing appears at the end of this paper in the section Self-description.)

Each event is specified in the compiler code by a DEB_EVENT macro. This macro takes one or more parameters. The first parameter is the name of the event, and subsequent parameters specify additional code that causes instrumentation data to be displayed.

An event will not occur unless its name has been given either on the command line that invoked the compiler or via a simple interpreter that is linked into the compiler. The interpreter displays event names and allows breakpoints to be set or canceled on particular events. For example, the Ada compiler implements a sophisticated syntax error recovery scheme that attempts a large variety of local corrections when an error is detected. When the parser makes an unexpected correction, events in the recovery code can be set to gather the data to determine why. Events in the recovery code are set by the setting of breakpoints on all events whose names start with PAR_RECOVERY. The result is an informative display at the start of error recovery, and another display as each kind of recovery is attempted. The displays can then be used to determine the reason for the particular recovery chosen.

The information obtained by setting an event gives precise information that is needed to determine why the compiler code made a particular decision, as opposed to the more general information given by the VMS Debugger. Often the time saved in analyzing each problem exceeds the amount of time required initially to put the events into the code. Furthermore, such events are still in place for the benefit of future developers who need to make enhancements or debug other problems.

Self-checking

As mentioned previously, the VAX Ada product contains approximately 500,000 lines of BLISS source code. Of these lines, approximately 5 percent are concerned with consistency checking (self-checking) of some kind. This is not very much incremental code in terms of overall development cost, yet the reliability and productivity benefits have been enormous.

The following sections examine some of the consistency checks we incorporated in the VAX Ada compiler for use by developers and maintainers. We look at the use of assertions in the code, at the use of special macros to mark unimplemented features, and also at how we used self-checking to track down memory-management errors.

Assertions in the Source Code

An old idea in software engineering is to include assertions in the source code. In its simplest form, an assertion is a simple expression whose value should be true at a given point in the code. If the assertion is false, then something is wrong and execution should be aborted. Although the idea of assertions is not new, we believe that their value is underestimated and that assertions are too often neglected in developing large software applications.

Detecting an internal error — often well before the error leads to a compiler crash or, worse, bad code is generated — is the primary advantage of an assertion. Assertions often point out errors that otherwise would not be noticed during internal testing.

Assertions also help in analyzing failures, as they provide a very good point at which to start a search for the cause of an error. In a complex, multiphase compiler, a bug in an early phase can result in a compiler crash in a much later phase or in the generation of bad code. In many cases, the relationship between symptoms reported by a user and the actual problem can be very remote and obscure. For example, approximately half of all performance failures reported by users of the VAX Ada compiler trigger some kind of assertion failure when compiled using the debugging version of the compiler. As a result, many problems that might have required days to fix in the absence of assertion checks have been fixed very quickly because we knew where to look for the problem. Although we have no statistics, we have no doubt that assertions have saved an enormous number of maintenance hours.

Assertions also help in day-to-day development, debugging, and project management. Simple inspection of the assertion failure message is enough to know who should be the first to look at the problem, and the person assigned to investigate the problem has a good idea of where to look. Assertions are also useful when the code is enhanced, as new code is checked against the assumptions made by the original programmer.

We implemented assertion checks using a series of BLISS macros. (Although BLISS macros were used to implement the checks, similar effects can be achieved with subprograms in other languages, such as VAX Ada, if the compiler evaluates static, constant expressions at compile-time and supports inline expansion of subprogram calls.) These macros are listed in Table 1.

Each macro takes two or more parameters. The first parameter is the assertion (expression) to be checked. The second is a text string to be displayed in the diagnostic produced when the assertion is false. By convention, this text string includes the name of the routine in which the failure occurred in order to simplify assigning initial responsibility (blame) for the failure. Any additional parameters are interpreted as additional code to be executed if the assertion fails;

Table 1 Assertion Macros and Their Effects

Macro Name	Effect in Debugging Compiler	Effect in Production Compiler
DEB_ASSERT	If assertion is false, then give a diagnostic message and enter VMS Debugger.	None
DEB_WARN	If assertion is false, then give a diagnostic message and continue.	None
DIAG_ASSERT	Same as DEB_ASSERT.	If assertion is false, then abort compiler.

typically these parameters are used to display additional information related to the failure.

Numerous assertions in the source code can have a negative effect on performance. For example, the consistency checks in the Ada compiler increase compilation time by about 50 percent. Thus, if assertions are to be included in the final product, developers will naturally hesitate to use them freely. We addressed this problem by causing the DEB_ASSERT and DEB_WARN macros to be conditionally compiled. The assertion checks are made only in a debugging version of the compiler that is used for internal testing. The macros are compiled as "no operations" in the production version of the Ada compiler and thus have no impact on performance.

On the other hand, it is desirable in some situations to retain the self-check in a production version but cause a failure to behave differently than it does in the debugging version of the compiler. The DIAG_ASSERT macro addresses this situation. DIAG_ASSERT behaves in the same manner as the DEB_ASSERT macro in a debugging version of the compiler; however, DIAG_ASSERT aborts a production version of the VAX Ada compiler. (The abort reports failure of an internal consistency check and requests that the user submit a problem report.)

These three assertions — DEB_ASSERT, DEB_WARN, and DIAG_ASSERT — are the most common form of consistency checking used in the VAX Ada compiler. More general kinds of checking are provided, for example, by specialized analysis routines and even complete traversals over the in-memory tree.

Marking Code Paths for Unimplemented Features

We adopted a rule during the development of VAX Ada that the software at each intermediate base level had to be robust. We required that the compiler diagnose the use of an unimplemented feature rather than crash or generate bad code.

This form of self-checking was implemented by the two macros called DIAG_NYI and DIAG_NYI_STOP. These macros are called with a text string that identifies the particular feature that has not been implemented. The execution of either results in a "not-yet-implemented" diagnostic. DIAG_NYI is used in situations where processing can continue after the diagnostic. DIAG_NYI_STOP is used to indicate that the compiler should be aborted after reporting the problem since there is no easy way to recover gracefully.

These macros proved to be a good clerical device for keeping track of work remaining. In addition, our approach — never leave a hole — contributed greatly to the reliability of the product. Robustness was the norm throughout development rather than a last-minute, clean-up activity. Over a long development effort, it is easy to put off writing a particular code path for another day and even easier to forget about it as the days and months pass.

Tracking Memory-Management Errors

The last approach to self-checking we discuss in this section is the use of special consistency checks to help track down some obscure memory-management errors in the compiler. Memory-management problems can be very difficult to diagnose because, for example, large programs often operate correctly for a long time after a routine writes to the wrong location in memory.

The error-tracking progression that we describe here occurred during the development of the initial version of the compiler. In each of the three problems in the progression, the introduction of a new check led immediately to the discovery of additional cases where the same error was occurring but, for whatever reasons, no negative consequences had yet been observed. Each of these cases was a bug that would eventually have been triggered, requiring many hours of a developer's time to debug. Finding the errors as a result of one of these checks was far less expensive in terms of development time than finding them one at a time as each situation arose.

The first problem we discovered in the progression was that a block of memory was being freed as expected, but the block size specified in the tree was larger than the amount of memory originally allocated for the block. To guard against this behavior, we allocated (in the debugging version of the compiler) an extra longword for each request and used it to remember the allocated size. This procedure allowed us to check the deallocation requests.

Later, we discovered that a routine was attempting to deallocate a block of storage back to a zone (subheap) other than the zone from which the block was allocated. We coped with this behavior by changing the extra longword to contain the Exclusive Or of the allocation size and the address of the zone control block.

Still later, we discovered that storage was being read after it had been deallocated. To cope with this behavior, we changed the deallocation procedure so that it overwrote the deallocated storage with all one bits. The one bits allowed us to distinguish unallocated storage ones from newly allocated storage, which is generally initialized to all zero bits.

Self-description

The primary data structure used throughout the compiler is a tree representation of the unit being compiled. This representation was made self-describing in order to

- Automate key algorithms in the compiler
- Simplify creation of internal consistency checks (self-checking)
- Simplify creation of some kinds of instrumentation
- Provide sophisticated debugging aids

Each node in the tree contains an eight-bit field named the KIND field. This field contains a value indicating the kind of information represented in that node. More than 230 kinds of nodes are used throughout the compiler. (There are many kinds because the tree is used not only for statements and expressions, as is common in many compilers, but also for declarations, in place of a more traditional symbol table. Indeed, except for comments, the entire unit being compiled is represented by a single tree.)

The KIND field is located at the same offset in every node; given the address of a node, it is easy to determine its kind and thus the information available in that node.

Moreover, the KIND field can be used to access a "node property table" in the compiler that contains a description of the fields in each kind of node, including the fields' types, offsets, and so on. Because each node describes itself in its KIND field and because the KIND field can be used to access the node property table, we refer to the compiler tree as a "self-describing" data structure.

The source-code definition of the tree representation can be thought of as essentially a variant-record type, where the kind value is a tag that discriminates among the variants. The actual run-time description of the tree representation goes beyond the level of detail that can be expressed even in a strongly typed language, such as Ada. For example, the description distinguishes between four kinds of pointer fields — all of which are simply pointers to other nodes in the tree from a data-type point of view. However, it is the presence of the variant-record definition itself as part of the compiler that is unusual and leads to valuable implementation techniques.

Automation of Key Algorithms

Several parts of the compiler use the node property table as a major part of their operation. For example, the part of the compiler that reads and writes the tree representation to disk, called the compilation library component, is driven almost completely from the node property table. As a result, we can easily add, delete, or change a field, introduce new node kinds, and so on. After a change is made, all that is needed is to recompile the few BLISS modules that create the node property table, link a new compiler, and continue development. The compilation library code does not need to be recompiled, let alone modified, to reflect the change; it adapts automatically.

Similar considerations apply to other parts of the compiler. In particular, the compiler has an algorithm for copying trees that is fundamental to the implementation of generic instantiation, inline expansion of subprogram calls, and default parameter evaluation. This algorithm is also heavily driven in part by the node property table.

Many utility routines also make good use of the node property table, for example to create a node of a given kind — given the code for the kind — the required size is obtained from the node property table, and each field of the new node is properly initialized as appropriate for that type of field.

Self-checking Based on the Node Property Table

We have described some kinds of self-checking earlier in this paper; it is also interesting to see that some self-checking is based on the node property table. First, we must back up and be a little more precise in our vocabulary.

Although we talk of the "tree," this is really something of a misnomer. The tree is really a general directed graph. However, there is a subset of the pointer fields that, in fact, does determine a spanning tree — a set of paths that spreads from the root (the COMPIL_UNIT kind of node) and reaches every node exactly once (and without cycles). Pointer fields that define the spanning tree are called "son pointers," whereas all other pointer fields are called "attribute pointers." (Son pointers are one of the several kinds of pointers alluded to earlier; there are three kinds of attribute pointers.)

Because the "tree-ness" of the program representation is so important to the correct operation of the compiler, one of the most important self-checks the compiler makes is to ensure that the tree really is a tree. This self-check is accomplished by a routine that starts at the root (the COMPIL_UNIT node) and uses the node property table to visit every node in the unit. Every son pointer is followed. As each new node is encountered, a bit is set. (This bit is reserved at the same fixed position in every kind of node.) If a node is encountered that already has the bit set, then a cycle has been detected (and a bug exposed). This check is performed at least twice, sometimes three times, during a compilation when the debugging version of the VAX Ada compiler is used.

Instrumentation through the Node Property Table

The node property table also provides a valuable tool for implementing certain kinds of instrumentation. Statistics on kinds and amount of storage by kind are readily calculated using simple tree walks like the one described for self-checking in the preceding section.

Enhanced Debugging

Finally, the node property table provides the basis for the variety of debugging display routines that were written as part of the project. As described in the section Instrumentation, these routines go well beyond what could be achieved by even the best general-purpose debugger, including the VMS Debugger. Rather than show the tree as a series of independent nodes, we can display the tree as the nested data structure it really is. Extraneous information, such as the addresses of nodes that are not legitimately pointed to by attribute pointers, can be suppressed. Certain kinds of nodes that are actually "private" in the Ada sense can be displayed in a natural manner by display routines created as part of the implementation of these abstract node kinds.

Figure 1 illustrates one such display for a simple example program. It is not necessary to understand this display in detail; rather, compare the kind of display one could get from node-by-node displays versus the highly annotated and interpretive example shown. A display tool such as the one that produced this example clearly is application specific and could be produced only as part of the project in which it is used. More importantly, even a project-specific tool such as this would not be practical without the run-time self-description of the data structures in use.

Summary

During the development of VAX Ada, we relied on established design and implementation methodologies, and we made extensive use of VMS tools

```
     1   procedure FOO is
     2       X : INTEGER := 0;
     3   begin
     4       if X > 0 then X := X - 1; end if;
     5   end;

0088D3B0:
     FOO ~ K_PROC_BODY_DECL<ANA_USED @1:0
         <<void>>:
         <<void>>:         _CORRES_BLOCK
         K_BODY<
008AD9F4:    K_DECLS< @2:4
008AD898:'      X ~ K_OBJ_VARIABLE<NON_CONST, ANA_USED @2:4
            .       INTEGER ~ K_REFER<STATIC 0000D230: @STANDARD/4>
            .       0 ~ K_INTEGER_VAL<CTC_VAL, STATIC @2:22
            .           SL>
            .       CONF_BEG_SEQ     _OBJ_FLAG1
            .       OCCURS_IN_NAME           _OBJ_FLAG2
            .       008ADDE0:        _SYMTAB
            .       <null list>      _PRAG_REP_CC>
            .   <no flags set>       _DECLS_FLAGS
            .   <<void>>:            _CONTINUE
            .   00000000   _CL_VIS>
              <
            .   K_IF_STMT< @4:4
            .       K_BINARY_OP<NON_CONST @4:9
            .           GT_INT
            .           X ~ K_REFER<NON_CONST 008AD898: @2:4>
            .           0 ~ K_INTEGER_VAL<CTC_VAL, STATIC @4:11
            .               SL>
            .           0000D230: INTEGER @STANDARD/4  _BINDING
            .           0000D140: BOOLEAN @STANDARD/2  _RES_TYP>
            .       <
            .           K_ASSIGN_STMT< @4:20
            .               X ~ K_REFER<NON_CONST 008AD898: @2:4>
            .               K_BINARY_OP<NON_CONST @4:25
            .             .     BINARY_MINUS_INT
            .             .     X ~ K_REFER<NON_CONST 008AD898: @2:4>
            .             .     1 ~ K_INTEGER_VAL<CTC_VAL, STATIC @4:27
            .             .         SL>
            .             .     0000D230: INTEGER @STANDARD/4  _BINDING
            .             .     0000D230: INTEGER @STANDARD/4  _RES_TYP>
            .               <no flags set>    _ASSIGN_FLAGS>
            .       <void list>
            .       <null locator>          _LAST_LOCATOR

            .       <no flags set>          _IF_FLAGS>>
             <void list>
             <<void>>:
             NO_EXCP_PART            _BODY_FLAGS
             <null list>   _PRAG_REP_CC
             FF7529B8:     _ZONE
             @3:0          _BEGIN_LOCATOR
             @5:0          _LAST_LOCATOR
             0088CA1C:     _SIGARGS
             TIME          _SAVED_OPT_STATE
             TIME          _LOCAL_OPT_STATE
             <null list>   _LOCAL_SUPP_CC>
         DST_HAS_SEG, DST_HAS_ZEM, EXIM_ALLOWED, IS_ELABORATED, IS_GBL_VIS,
           IS_LIB_UNIT, MECH_FIXED        _PROC_FLAGS
         <null list>                _PRAG_REP_CC
         0088CA1C:         _SYMTAB
         <<void>>:         _FULFILLS
         <<void>>:         _STATUS_OBJ>>
```

Figure 1 Example Tree Display in the VAX Ada Compiler

(VMS Debugger, VAX Performance and Coverage Analyzer, and so on). However, we also used some relatively simple internal tools and accompanying development philosophies to help us increase our productivity, improve the reliability of our product, and decrease maintenance costs.

Automation, instrumentation, self-checking, and internal self-description all played major roles in our day-to-day practices, from early design phases through field test. We continue to use these tools and techniques in the ongoing maintenance and evolution of VAX Ada. We hope that our successful experience with these pragmatics on the VAX Ada project will help promote wider interest in the use of such ideas on other software projects.

Steven J. Grass

Development of a Graphical Program Generator

To develop an unprecedented graphical-interface product for generation of COBOL applications, project engineers explored a new development approach. During the advanced development phase, types of generators were researched, a prototype was built, and product goals were outlined. In the product development phase, the major components of the VAX COBOL GENERATOR software — the data dictionary, the work-file system, and the graphical display — were designed and coded. In addition, developers integrated existing components into the generator. Testing, design documentation, and project review were also part of this phase. This development approach, combined with the use of several development tools, proved to be productive and resulted in a stable and reliable product.

The VAX COBOL GENERATOR software is a fourth-generation language approach to the creation of commercial applications. Using this generator, the programmer draws a picture resembling a flowchart of the final application rather than use an editor to write lines of code. This picture produces VAX COBOL code, which can be compiled, linked, run, and debugged. All maintenance is performed at the graphical level.

This paper describes the development of the VAX COBOL GENERATOR software from initial concept to product shipment, a process that took approximately three years. Because this was the first product containing a graphical interface developed at Digital, many unique project development problems were encountered. This paper discusses how the project team solved these problems in the research, development, document, test, and project review stages.

The project can be divided into two major phases. The first phase involved advanced development and lasted one year. During this phase, a prototype was developed and demonstrated to various management groups.

The second phase was product development and lasted two years. This product development phase was divided into two major base levels, that is, milestones at which specified capabilities are complete. At the first base level, a skeleton was built that contained most of the core functionality of the generator. Once the skeleton was completed, additional functionality could easily be added. The skeleton consisted of the work-file system, some screen interface routines, the driver for the main screen editor, and the driver for the code generator. Each member of the team was responsible for one of these components of the skeleton. The second base level marked the enhancements to the skeleton and the definition of system functionality in design documents.

Advanced Development

A program generator was so unlike other products being designed at Digital at that time that it was necessary to spend the first project phase on advanced development work. During this time, research could be performed, ideas could be exchanged between the developers, and breadboards and prototypes could be created. This advanced development phase indeed proved to be worthwhile and a significant step toward the product's success. In particular, the development of the prototype provided a way to communicate concepts to each other and to management. Remarkably, the prototype, although crude, incorporated all the underlying concepts contained in the final product.

From June 1983 to June 1984, two developers worked on the advanced development of the VAX COBOL GENERATOR software. This section discusses that work, including the creation and demonstrations of the prototype.

Defining Product Specifications and Beginning Research

The specifications for the proposed product were open ended. The one, general product requirement was that the program generator would generate VAX COBOL code. At that time, the kind of generator to build, the interface to use, and other aspects of the product were not yet understood or specified.

The first two tasks facing the developers, therefore, were to determine what a program generator was and what currently existed in the marketplace. They learned that although much had been accomplished in the development of fourth-generation languages, not all product approaches had been fully explored. They saw the advantage in creating a type of generator that not only increased programming efficiency but also was simple and interesting to use.

They learned there are two types of generators: application generators and program generators. Each has its own advantages and serves a distinct market.

An application generator processes commands interpretively. It does not produce source code. The application generator has a close relationship with its application's database and is used mainly for relatively simple data retrieval and report generation. Where execution speed is not critical, programmers use application generators for quick development turnaround. DATATRIEVE, RALLY, and Cognos' PowerHouse are examples of application generators.

A program generator, on the other hand, produces source code. Program generators can generate anything from BASIC to Ada program code. Because the code produced can be compiled, execution speed of the created application is faster than the execution speed of a similar application produced by an application generator. A program generator does, however, take longer to develop the application.

Most program generators produce about 70 percent of the final application. However, the applications produced are skeletal and have to be edited after development. The generated programs consist of the high-level structure, but many of the lower level routines still need to be edited by hand. Once having edited the generated program, the developer can no longer enhance the product using the generator since the hand-coded changes would be lost. Therefore, after a program has been developed, all program maintenance must be performed at the code level. The development time saved using the program generator is only with reference to the program development phase, not the program maintenance phase.

The VAX COBOL GENERATOR team saw an opportunity to close the gap between program development and maintenance. They decided to produce a program generator that would create the entire program. Software development gains in terms of developer time saved would then extend beyond development phase to encompass the maintenance phase as well.

The team also saw another opportunity. Most existing generators are restricted in the types of applications that can be generated. Although generators could relatively easily create typical commercial applications, complex applications were more difficult to create. The generator team decided to build an open-ended product that stressed flexibility in the level of program complexity.

Finally, most of the interfaces of the generators we studied were menu-driven. Users were required to repeat continually the same steps in order to create the application. The generator team felt that a user-friendly human interface would be a more expedient tool and more appealing to users as well.

The decision to produce a graphical interface for the generator was one of the first the development team made. In 1983 the VT200-series terminals were beginning to be shipped, and graphics workstations were starting to be developed. It was evident to the generator team that graphical terminals would become integral to program development work.

Moreover, human-factors research and the developers' own experience with graphical workstations confirmed the decision. During research, the developers spent a great deal of time using the revolutionary Xerox Star workstation, which had been introduced in the early 1980s.

This workstation demonstrated the power that windowing and icons can give to software development. The developers felt that they could expand the concepts of the Xerox Star (later further demonstrated in Apple's MacIntosh computer) to the area of program generation. The icons could represent the various data and procedural entities in typical programs, and the windows could be used to define these entities. The flexibility of a graphical workstation allowed this to happen.

In summary, research into types of generators and user interfaces helped to determine the following product goals prior to the development of the prototype:

- The generator would produce an entire VAX COBOL program, thus extending use of the generator through the maintenance phase.
- The generator would have the flexibility to create complex as well as simple types of applications.
- The generator interface would be a graphical interface; it would be easier and faster to use than conventional editors.

Project Value of the Prototype

In the early research stages, the product ideas formulated by developers were so unlike any previously developed products that the ideas were difficult to explain and demonstrate. Icons, for example, were not in general use at that time; and the ideas of boxes and lines representing operations and control flow were entirely new concepts and were difficult to grasp. Attempts to draw the ideas on paper failed since drawings could not show the facile action of the interface that the developers visualized. Developers decided they needed to construct a prototype.

Creating a prototype before specifications were clearly defined was a risk the developers wanted to take. The simple prototype proved them right. It was invaluable for communicating project concepts among the developers and to management.

Their first task was to decide what functions to build into the prototype. A prototype is written to demonstrate ideas, without regard to maintenance or the performance of any programs produced. Therefore, the developers knew that the prototype code would have to be discarded when the product development began. In order to ensure that no code would be reused, it was decided that the entire prototype would be written in the VAX COBOL language. The final product would be written in the preferred development language, VAX BLISS. Also, it was decided that a graphical software package such as the Graphical Kernel System (GKS) would not be used for the prototype. The strengths of GKS were terminal independence, easier maintenance, and better performance, none of which were goals of the prototype. In addition, the time taken to learn GKS would cause a needless slowdown in the prototype development. Instead, ReGIS escape sequences would be output. This decision to output ReGIS directly instead of using GKS would speed the effort to obtain a working prototype.

As it turned out, many major functions of a program generator were completely defined within the prototype. The method of form, report, and file definition, as well as the concepts of procedural and data flow could all be visualized using the prototype, which even did program generation. The prototype was so complete that a member of Digital's Management Information Systems (MIS) department used it to develop applications to be used within his department. The prototype demonstrated how these high-level graphical concepts could be translated into the generation of source programs.

Product Specification Approval

Demonstrations of the prototype were given well over one hundred times to all levels of management, project leaders, some customers, and Digital's Research and Development Committee. Reaction was positive. It was agreed that the product was ready for development. Unfortunately, reviewers had little with which to compare the product and were therefore unable to offer the constructive criticism the developers were seeking. Feedback on the product would be gained through the more painful process of experience.

Product Development

The product development phase of the VAX COBOL GENERATOR software started in June 1984 and ended in September 1986. This two-year period began with the first written design specifications and closed with delivery of the product. Product development included design, coding, and testing phases. No more than four software engineers were assigned to the project at any one time. An additional three engineers were assigned to write the graphical package (described later in this paper). The final product had over 140,000 lines of source code, or nearly 3000 lines per developer per month during the coding phase. This achievement was considerable when compared to 1985, when 650 lines of code was the average number written per developer per month.[1] The productivity of the VAX COBOL GENERATOR team was due in large part to Digital's software development environment tools.[2] Some of these tools are described later in this paper.

This section gives a brief overview of the COBOL generator product. Following this overview are descriptions of the components and systems selected for and designed in the product's implementation. The major components of the generator are the data dictionary, the work-file system, and the graphical display system. Also described in this section is the reuse in the generator of previously developed components.

Product Functions Overview

Programs are graphically described within the VAX COBOL GENERATOR product by a combination of nodes and connections between these nodes. After the nodes and connections are created and defined, the VAX COBOL GENERATOR software can create a VAX COBOL source program which can be compiled.

A node graphically represents data and operations to be performed on that data. There are eight node types, and they fall into two categories: procedural and data. Procedural nodes represent functional tasks to be performed in the program or represent structure in the program. Examples of procedural nodes are those that represent the movement of data between two data nodes, a sorting operation, or the manipulation of a menu. Data nodes represent data to be accessed in the program. Examples of data nodes are forms, files, and reports.

The connections between the nodes represent flow in the program. Procedural connections show procedural flow; data connections show data flow.

The programmer creates nodes on each level of an application and connects them to show procedural and data flow. Editors, pop-up forms, and pop-up menus prompt the programmer for detail about the nodes and connections. From this information, the VAX COBOL GENERATOR software creates a VAX COBOL program.

Figure 1 shows an example of a VAX COBOL GENERATOR screen. The procedural type nodes READ-INFO and SHOW-ERROR are shown as are the form node EMP-FORM and file node EMP-FILE. The data connections are shown as dashed lines, and the direction of the connections indicates that data is to be read from the form and written into the file. The procedural connection is shown as a solid line that indicates control

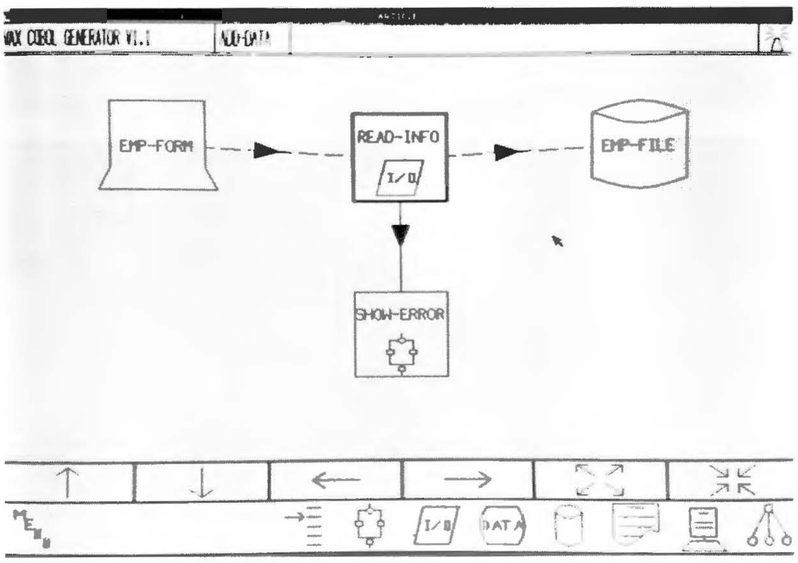


Figure 1 VAX COBOL GENERATOR

flow is to go from the READ-INFO data movement node to the SHOW-ERROR procedure node.

The generator helps the novice user via several functions. At any time, the programmer can use the Help key to obtain context-sensitive information about an operation. The programmer can also use the HELP command or choose help from the menu. Easy-to-understand error messages also guide the programmer through the design process.

The VAX COBOL GENERATOR software enforces top-down programming. The programmer begins program definition at the top level of the program. Group nodes, which represent much more complex operations at a lower level, create structure within the generated program. Editing a group node moves the programmer down one level in the program, thus breaking the program into smaller, more modular pieces.

Other VAX COBOL GENERATOR features include data dictionaries and libraries for the storage and reuse of common data and procedures, an Rdb/VMS interface, a complete journaling capability, and a method to escape into an editor where user-defined COBOL code can be entered into the generated program. Also available are program documentation facilities that include a map with a breakdown structure of the program. These facilities also permit the addition of user-written documentation to parts of the program.

The Data Dictionary Asset

A main component of the VAX COBOL GENERATOR is its data dictionary. Although this component was not included in the prototype, the developers found in later research that nearly all fourth-generation languages, no matter how primitive, contain a data dictionary. As Digital's MIS department pointed out, a successful product must contain a depository for reusable programming. Developers therefore devised a method by which users of the VAX COBOL GENERATOR product could easily define data and procedural entities in a central location. Users could then share these entities within one or more programs developed using the generator.

The VAX Common Data Dictionary (CDD) was the current Digital standard for sharing data among the layered products. CDD is excellent as a standard for sharing record-structure definitions. Developers, however, needed a method that could in addition understand entities defined by the user within the generator, such as reports and user-defined procedures. Therefore, the CDD was used within the generator so that users could optionally share record definitions; another method, the generator library, was devised so they could share other components of the generator.

The generator library lets the user share any form, file, report, local storage, procedure, or field definitions. If a user of the generator wants to share a program component, this component can be either defined directly in the library or stored in the library from an application. Any other program developer wishing to reuse that component in another program can simply reference it.

Each time a new node or field definition is created within an application, the generator performs a search through all known generator libraries. If a match is found, the user is given an option of referencing the component from the library. If he chooses to reference the component, the previously defined component is then read into the application and is thus reused. The internal representation stored in a library file is identical to its counterpart defined in the application. Because of this, no conversion is required to be performed by the generator when a component is referenced, and the internals of the generator do not need to know whether the component is referenced from a library or is defined within the application.

The data dictionary utility proved to be one of the strongest assets of the product for two main reasons. First, because components can be reused in many applications, users' programs can, for example, all have the same interface. Each user does not have to redefine the components for his particular application. Second, multiple users can simultaneously develop different pieces of the same program. Each user can define components in a different library, then one user can integrate these components into the application.

The one restriction is that only the generator can read from or write to the library structure, because the library structure is defined by the VAX COBOL GENERATOR software. Forms can be shared among COBOL programs developed by the generator, but not with any other language or tool.

The Work-file System

Another key component in the VAX COBOL GENERATOR software is the work-file system structure, a key component in any software product. It was important to the product's success to give users the ability to quickly access the generator's database on disk and to manipulate records in memory. The developers were looking for an easy-to-use, efficient interface when they decided on the format of the VAX COBOL GENERATOR database file and the associated manipulation routines. Instead of developing a new file structure, the developers decided that the database would be an RMS indexed file. They could then use standard VAX RMS file manipulation routines rather than write new routines. Time saved during development was considerable. Most work-file systems take months to develop; the generator's system was performing within two weeks. Run-time performance, thought at first to be a possible problem, was acceptable.

After the work-file records were read into memory, the routines used for manipulation were patterned directly after their RMS file counterparts. Each record contained a key, so records were read by key and then read sequentially. Records defining a node were logically grouped together since their keys were alike.

The Graphical Display System

Another key component in the VAX COBOL GENERATOR is the graphical display system. The VAX GKS program is Digital's implementation of the ISO (IS 7942) and ANSI (ANS X3.124-1985) GKS standard for two-dimensional, device-independent graphics. The VAX GKS program had just been released when development of version 1.0 of the VAX COBOL GENERATOR software began.[3] Because it appeared to contain all the graphical primitives and terminal independence for which the generator team had been looking, it was chosen to be the graphical system for the generator.

Early in the development cycle, however, it became apparent that a higher level graphical interface was needed. GKS provided the required functionality, but the routines were too low level for the developers' purposes. By layering a higher level interface above GKS, the generator's internals would be simpler and, therefore, easier to develop and maintain.

Using GKS calls, a node representation on the screen would be extremely complex to construct. The generator would have to make multiple calls to GKS routines to create the square containing the node, draw the text, and draw the icon representing the node type. In addition, calls would have to be made to determine if all or part of the node would be visible, to determine which font to use for the drawing of the text, to determine the select area for the node, and others as well. Instead, one higher level routine performs all these functions.

Another product being developed, the VAX Software Project Manager program, also needed a graphical interface with the same capabilities our product required. Consequently, the group of three engineers, mentioned earlier, developed the high-level graphical manipulation routines layered above the VAX GKS software. These routines would be completely terminal independent. Any product using these routines would run on a terminal, where the cursor is manipulated by arrow keys, or on a workstation, where the cursor is manipulated by a mouse. By sharing the same human-interface routines, both the generator and the VAX Software Project Manager program have the same appearance and interface. Consequently, users who learn one product's interface can more easily learn the other.

Because developers needed to focus only on developing the high-level screen interface, they were able to spend more time writing the generator internals as opposed to developing the graphical display system internals.

Use of Existing Components

As noted earlier, the implementation language for the VAX COBOL GENERATOR software is the VAX BLISS language. Because Digital's software products had been written in BLISS, any components written for these existing products could easily be integrated into the generator. Time to market was important, so the time-saving use of any already written software was encouraged. Consequently, during the design stage, developers decided to reuse some of these components.

In addition to saving schedule time, the use of these existing components meant greater product stability. The components had been used within Digital's products and therefore had been tested by customers for years, and the components would be tested again after integration in the generator. After the integration had been completed, the reused pieces contained fewer errors than any other components within the generator.

This section describes the components that were used or adapted within the generator.

Forms Editor

The definition of a form node required the use of a forms editor to define the layout of the form on the screen. Digital had two forms products at that time: VAX FMS (Forms Management System) and VAX TDMS (Terminal Data Management System) programs. Not only did each contain a forms editor, but the key definitions within each were identical. The VAX COBOL GENERATOR developers decided to use this established set of key sequences.

The similarity between the FMS and TDMS keypads is not a coincidence. Developers of TDMS modified the FMS sources for their TDMS product. The generator developers also decided to modify the FMS sources. Only two changes were needed: one was to change the field attributes such that they were particular to the generator, such as autoterminate; and the other was to include an interface to the generator's data dictionary. Routines were written to convert between the forms editor's internal data structures and those of the generator.

Within a very short time — one week — the forms editor had been integrated and was working within the generator. Writing a forms editor from scratch would have taken much longer, perhaps six to eight months.

File System

A set of routines was needed to access the product's various files, such as the generator database, generator libraries, journal file, and any COBOL files that would be generated.

To perform the standard set of operations on these files, the generator developers chose the file I/O system developed for the VAX TPU (Text Processing Utility) software. It contained the standard open, close, read, and write routines in a modular, easy-to-integrate form. As with the forms editor, integration was simple and stability was high.

CDD Interface Routines

The VAX COBOL compiler already contained routines that read in records defined within the VAX Common Data Dictionary software. The VAX COBOL GENERATOR developers converted these routines so that they recognize generator data structures. Again, the use of previously written interface routines saved a great deal of development time.

Design Documents for Project Use

The developers wrote a design document for each major piece of functionality needed for version 1.0 of the VAX COBOL GENERATOR software. These documents included the following:

- The reason for the new functionality
- A high-level description of the functionality
- Details of the functionality, including each routine to be written or enhanced
- A test strategy
- Resource and time estimates for design, coding, and testing of the new functionality

These documents had three purposes. The first purpose was to have the team review the proposed implementation of the new functionality. Any discrepancies in the graphical interface design or internal implementation were found early, before actual coding began. The second purpose was to give the project leader a reliable estimate of coding time for determining future schedules. The third purpose was to help the individual writing the documentation keep current on all new features. With this information, the writer could allocate the appropriate time needed to write about new features.

Product Testing — Internal and External

Three different types of testing were performed during development of the VAX COBOL GENERATOR software:

- Testing by the developers
- Human-factors testing, by the Software Usability Engineering Group
- Field testing, at Digital's internal sites and at external (customer) field test sites

The standard internal test method for software at Digital is the VAX DEC/Test Manager (DTM) software.[4] DTM can test generated data and text screens. DTM can test the generator database file, any libraries, and any generated COBOL source code. A large DTM test set consisting of approximately two hundred tests was run at periodic intervals and at base levels. Tests were required to be written for each new piece of functionality.

However, DTM has no capability for testing graphical screens. In fact, the developers discovered that there was no product available that could test graphical and asynchronous interfaces. Any previous graphical product was tested interactively by the users, and this method was also used for the generator.

At each base level, the VAX Performance and Coverage Analyzer (PCA) software was run on the entire test set to determine test coverage. If PCA determined that a large piece of code was not being tested by the test set, one of the developers wrote a new test for the code.

The second type of testing was human-factors testing, performed by the Software Usability Engineering Group.[5] Early in the development effort, test subjects (programmers) unfamiliar with the VAX COBOL GENERATOR software were asked to perform simple tasks using the product. The test report listed and prioritized all problems the subjects encountered. As a result of this report, changes were made to both the human interface and to the documentation.

For example, one problem, which accounted for the largest percentage — 19 percent — of the total error time, involved field termination in the editor used for the definition of record structures. In this editor, the Return key was used to create a new field's definition. Users were mistakenly using this key when attempting to move from one attribute to another for a field's definition. As a result, new field definitions were being created inadvertently. The generator team changed the definition of the Return key as a result of the human-factors testing, and selected another key for the creation of new fields.

Other changes were implemented as a result of this testing. Better labeling was devised within the generator, and better, easier-to-understand documentation and context-sensitive help messages were written.

The third type of testing was field test. This testing was divided into three phases. The first phase began in September 1985 and included an unlimited number of Digital sites and a limited number of customer sites. Practically all functionality was included in this release except for report and sort nodes. Sites were asked to test how well the VAX COBOL GENERATOR software fit into their development environments. Unfortunately, feedback from external sites was very limited in this phase.

The second phase began in February 1986 and marked the beginning of serious field test. The VAX COBOL GENERATOR software, with nearly all functionality, was installed in 14 customer sites and 200 Digital sites. Excellent feedback was given from all sites. One customer site was able to generate a 100,000-line program during this five-month period.

This site notified the developers immediately whenever a problem occurred; developers could then fix errors in the product before it was released to the public.

The third and final phase of field test began in July 1986. All bugs found in the earlier tests had been fixed, and all final functionality was included in the product. This phase was used to ensure that no major bugs remained in the generator.

In summary, all three types of testing contributed to the stability of the final product. Many bugs were found and corrected, the documentation was simplified, and the human interface was improved as a result of the various types of testing.

Project Test Communications

The unique and efficient communication medium for internal test queries and responses was VAX NOTES conferences.[6] The VAX NOTES network communications product functions as an electronic blackboard and lets users conduct online conferences or meetings. Using the VAX NOTES program, any user on Digital's engineering network could make suggestions and ask questions during development, and report problems during field test. Because the developers monitored the NOTES conference continually, users were given feedback quickly. Additionally, the VAX NOTES program allowed the number of test sites to expand. Developers had more time to work with more than the usual number of sites because the program not only facilitated communications between sites but also maintained a record of any communications.

For communications outside the corporation during field test, a hotline was installed. Developers took turns answering the calls. The immediate feedback allowed for quick problem resolution, contributing to a successful field test. Also, an on-line Quality Assurance Report (QAR) system was available to all field test sites. The sites could log on to the machine containing the system and comment on bugs or make suggestions.

Developers would periodically scan the QAR database.

Project Management Tools

During VAX COBOL GENERATOR development, developers kept resource status and task records on paper. Group meetings were the only formal mechanism for the exchange of status information. A tool was used for scheduling, but it did not provide the task tracking capabilities that were needed. Simultaneous with the development of the generator, the VAX Software Project Manager program was being developed. Early versions of this tool were used at the end of the generator's development cycle, and ts benefits were readily apparent. Before the tool was available, the scheduling done on paper contained errors. The VAX Software Project Manager program automated the process. The scheduling estimates made with this tool were found to be much more reliable than those done manually. Moreover, tasks are graphically displayed, making it easy to visualize a schedule and to test various schedule scenarios.

Project Review Meeting

Immediately upon release of the VAX COBOL GENERATOR software, a project review meeting was held. All developers presented their views on how to improve the process for building this product. Although the development process had gone well, many valid points were raised at this meeting. For example, the newer mem-bers of the team had found much of the code difficult to understand. The code would therefore be harder to maintain. As a result of this discussion, plans were put in place to develop coding standards for future versions of the product. Another point concerned informally made design decisions that went unrecorded and were often lost. It was decided at the meeting to create a NOTES conference where discussions and decisions internal to the group could be logged. Using this conference, future developers of the generator could easily reference the earlier decision-making process.

Team members found the meeting to be an extremely valuable exercise and will hold such meetings at the conclusion of all future product releases.

Conclusion

The development process of the VAX COBOL GENERATOR software was successful. An entirely new process for the generation of programs was devised, prototyped, written, and tested. Moreover, the process was completed in an amazingly short period of a little over three years.

Three main factors contributed to the high productivity of the VAX COBOL GENERATOR team and stability of the final product:

- An early prototype, through which ideas could be presented for debate
- The reuse of existing technology, so time was not spent doing work that had already been performed
- Various forms of testing, where the technological and human interface designs could be tested

Acknowledgments

I would like to thank the developers who worked on the VAX COBOL GENERATOR software, including David Tarbay, Leo Treggiari, John Ronan, Deb Bourquard, Andy D'Amore, Bill Fountas, and Rich Phillips. Also, I would like to acknowledge the contribution made by the members of the graphical display team, Vick Bennison, Jay Bolgatz, and Jeff Orthober.

References

1. H. Davis, "Measuring the Programmer's Productivity," *Electronic Engineering Manager* (February 1985): 44–48.

2. B. Beander, "VAX/VMS Software Development Environment," *Digital Technical Journal* (February 1988, this issue): 10–19.

3. B. Axtell, W. Clifford, J. Saltz, "Programmer Productivity Aspects of the VAX GKS and VAX PHIGS Products," *Digital Technical Journal* (February 1988, this issue): 62–70.

4. L. Ziman, M. Dickau, "Project Management of the VAX DEC/Test Manager Software Version 2.0," *Digital Technical Journal* (February 1988, this issue): 110–116.

5. M. Good, "Software Usability Engineering," *Digital Technical Journal* (February 1988, this issue): 125–133.

6. P. Gilbert, "Development of the VAX NOTES System," *Digital Technical Journal* (February 1988, this issue): 117–124.

Linda Ziman
Martin Dickau

Project Management of the VAX DEC/Test Manager Software Version 2.0

To produce a complex, major software version in less than one-year's time, the DEC/Test Manager team became the first at Digital to use all available VMS productivity tools. As part of their strategy to meet a shortened schedule and at the same time maintain quality, the team chose an iterative development approach. Throughout the phases of requirements analysis, specification and design, and implementation, the team took advantage of the many software tools available to streamline every aspect of the project from source code management to performance analysis. The tools used include VAX NOTES conferencing, the VAX Language-Sensitive Editor, VAX DEC/CMS software, and the VAX Performance and Coverage Analyzer.

As software development has increased in complexity, software engineers have sought products that help to increase their productivity as well as the quality of the software they produce.[1] At the same time, market pressure to deliver more software products has increased. Software productivity tools are essential elements in the engineering effort to meet the market need for the same or greater software functionality delivered in a shorter amount of time. It is not uncommon for the development cycle of a major version of a software product to be longer than one year in duration. The VAX DEC/Test Manager team was able to deliver a major version to market in less than seven months. This paper describes the development methodology and the productivity tools used at various software life-cycle stages to achieve this shortened time to market.

VAX DEC/Test Manager Product Overview

The VAX DEC/Test Manager software is a productivity tool that automates the regression testing of software during the development and maintenance phases. Regression testing ensures that modifications made to the software do not affect the previously tested execution of the program.

The DEC/Test Manager allows users to describe tests as a set of files, execution environmental aspects, and processing options. These descriptions are stored in a DEC/Test Manager-controlled directory, called a "library," and are easily accessed, tailored, and managed via DEC/Test Manager commands.

The core of a DEC/Test Manager test description is a user-supplied script which the DEC/Test Manager executes when the test is run. Use of a script and the environmental and processing control specified in a test description ensures that only changes in the software being tested can alter the results of a test run.

After test execution is completed, the DEC/Test Manager automatically compares the results of the run with a set of benchmark results. The comparison statistics are available through a simple command, and the DEC/Test Manager provides a set of functionality and commands for locating and examining the results of those tests that indicated some type of change from expected behavior. All file management is handled automatically.

The VAX DEC/Test Manager software version 1.0 was mainly a batch testing system. Pressure came both from the market and from internal Digital engineering groups to produce a follow-on version that would do more, one that would test graphical applications on character-cell terminals, such as a VT52 or VT100 terminal.

From the outset of the project, the development team realized the difficulty of the design problem. Adding to the difficulties of development would be the lack of a commercially available product with which to compare functionality for such a tool.

Project Setup

The DEC/Test Manager version 2.0 development team consisted of three junior engineers, a principal engineer, and a project leader. This team was to be responsible for producing the new functionality and for maintaining the previously released versions of the software.

At the beginning of the project, the team members recognized they would need to improve productivity in order to deliver a product in the time required. The work involved to produce version 2.0 would be as complex as it had been for the major version 1.0. Version 1.0 had taken 18 months to produce, but market pressure dictated a second version be delivered in 10 months or less.

Consequently, the DEC/Test Manager team was one of the first at Digital to use all of the VMS productivity tools currently available. The team's use of these tools is discussed in the sections below. The tools consisted of the following software:

- VAX Language-Sensitive Editor
- VAX DEC/CMS (Code Management System)
- VAX DEC/Test Manager
- VAX DEC/MMS (Module Management System)
- VAX Performance and Coverage Analyzer
- Problem Tracking or QAR system
- VAX NOTES
- Digital Standard Runoff

The project team also decided to use an iterative development approach. This approach differs from the development method of completing all requirements of one stage, or phase, in the software life cycle before proceeding to the next.[2] Instead, an iterative process allows problems to be detected, fixed, and quickly incorporated at any stage. Consequently, errors made in the design are caught and corrected long before the field test. Additionally, this approach allows the corrected software to be made available to the people who had detected the original problems. These users can then further evaluate and test the software.

The first phase in iterative development is product requirements analysis, which we discuss next.

Requirements Analysis Phase

One of the most difficult tasks of software engineering is forming a clear statement of the problems the software must solve.

Traditionally, on small, less rigorously structured projects, programmers may interview users about their needs. On large-scale projects, such as those done by Digital Software Engineering, more formal requirements analysis is undertaken early in the project's life cycle. However, no matter how formal the analysis process, the quality of the resulting problem statement strongly depends on the information gathered during that process.

The DEC/Test Manager version 2.0 team wanted to decrease the time normally taken to gather this information without decreasing the quality of the requirements analysis phase and without reducing the number of target markets contacted for information.

Also to be considered in gathering requirements information is whether the proposed product is new or a revised version of an existing product. For a new product, requirements are often based on the abilities of similar products or simply on programmers' and potential users' wishes. The requirements list for a new version of an existing product, however, can be an ex-panded and more precise list. Members of a user community can make suggestions based on their judgments of the deficiencies of a product. The problem facing developers is finding a mechanism by which to obtain this feedback in a short period of time from these users.

The team chose not one but several ways to obtain responses from widely dispersed and varied groups of users. These methods are the subject of the balance of this section.

VAX NOTES Conferencing

The DEC/Test Manager version 2.0 team used a variety of methods to quickly gather feedback on version 1.0 from groups in such diverse locations as Japan, England, and California. To obtain input from groups in similarly widespread locales, the developers of version 1.0 of the DEC/Test Manager had spent a great deal of time in the requirements gathering stage.

Significant time was saved through the use of VAX NOTES software. NOTES is a computer conferencing tool which the DEC/Test Manager team used to set up a forum for discussing its product.[3] Although NOTES at that time existed only as several prototype tools, it provided a workable forum for the entire internal user base. Not only was the team able to gather requests and ideas from the internal base, but it also made available preliminary specifications and gathered feedback before any prototyping was started.

NOTES alone, however, was inadequate for gathering all the feedback needed. At that time, only a small portion of the user community actively participated in the NOTES conference. Moreover, any feedback from just one source might be skewed. To correct for this possibility, a more direct form of input gathering was also implemented. The DEC/Test Manager versions 1.0 and 1.1 installation command procedures had been made to send VAX Mail notification of any Digital internal site installation to the DEC/Test Manager development account.

These notifications provided the team with a fairly complete list of the version 1.0 installed base. The team used the list as a distribution and interest list for sending users questionnaires and requirements-input requests. These requests served to draw more users into the requirements-analysis process and to raise the quality of the requirements analysis itself.

Quality Assurance Reports

Most Digital software engineering groups keep track of all bug reports and suggestions in Quality Assurance Report (QAR) databases. These databases are used to record problems, assign problems to specific developers, analyze quality statistics, and generally ensure that problems do not go unnoticed. Many groups, including the DEC/Test Manager project team, allow direct access to their QAR databases by all users, both external field test sites and internal users. Indeed, a QAR database had been used to gather feedback about versions 1.0 and 1.1 from external field test sites and was always available to internal users for reporting bugs or submitting suggestions. Therefore, the DEC/Test Manager team was able to cull product requirements from the internal QAR system tool.

Developer as User

The team itself took advantage of the DEC/Test Manager environment to gather requirements input and to analyze the input. The team felt that software developers who use their own software products produce higher quality software. Although software developers are already intimately familiar with the internal technical details of their products, if they are not also the *users*, they lack external perspective.

External perspective is the perspective of the customer or software user. A software developer who has this perspective can more easily understand and identify with customers' feedback. The DEC/Test Manager had always been used to test itself; therefore, the developers themselves were users of the product, were familiar with the tool's inadequacies, and knew where improvements could be made. As users, they were also able to screen requirements and specification ideas for usefulness before going to the user base for more suggestions.

Software Quality Management

Quality assurance groups are not common in Digital Software Engineering because Digital's management believes each software developer and each development group must be responsible for the quality of its own product. Although this is indeed the case, the Software Quality Management (SQM) Group performs a unique, cross-product quality assurance function. The VMS SQM Group, part of the VMS Operating System Group, works with the various product groups to ensure that all VMS products follow certain conventions for installability and inter-operability. The SQM Group therefore obtains a set of tests suites for each product and, on a frequent basis, tests how well products work with each other on the VMS operating system.

Through this process, the SQM Group provided valuable input to the requirements phase of the DEC/Test Manager development cycle. SQM required all product groups to submit regression test subsets that could be executed under the DEC/Test Manager software. This request led not just to requirements from the groups responsible for applications that run on the VMS operating system, but also to requirements from the SQM test administrators who needed test-control functionality to make their task easier.

In summary, through the combined use of these tools and improved information-gathering processes, the DEC/Test Manager team was able to move into the specification stage within the first weeks of the project. The time savings was significant since several months was the generally expected timeframe at this stage for a project of this scope and complexity.

Specification and Design Phase

During the specification and design stage, developers define what the system will do, how it will be used, and how it will be implemented.[1] This stage is often the longest in the development life cycle. Again the team sought ways to decrease the time required to complete the phase. They started by streamlining how the specification itself would be documented.

Since the DEC/Test Manager team did not know what method they were going to use to test graphical applications on character cell terminals, they wanted to explore a number of different design possibilities. They also wanted to ensure that as they refined their ideas the product specification document would be continually updated to reflect current specifications. They did not want to write the specification at the end of the phase when there would be a higher possibility of inadvertently leaving out details. As a result, they developed procedures that would allow for continual update of the specification and also enable them to prototype a number of alternative implementations.

Next the version 2.0 functionality was divided into major components, and each component was assigned to a developer for specification. A language-sensitive editor, VAX LSE, template was written both for specifications and designs to enforce a convention for information. As first drafts of specifications were completed, the team met to review the specifications and suggest changes. The changes were incorporated into the documents, and the next pass of the complete specification was built. (The main specification document was a Digital Standard Runoff procedure with Include files for each of the component files the individual developers wrote. It was a simple matter of processing to generate the latest specification, and as a result, processing was done frequently during this stage.)

Finally, each new specification was made available in a public directory for internal review. This availability not only enabled the entire team to be up to date on the current thinking for all version 2.0 components but also assured the document was continually reviewed, which helped the team complete the specification faster.

Concurrent Prototyping and Maintenance

A conflict presented itself at this point in the development process. Each developer wanted to test various creative solutions, and each developer needed to access any module in the system to quickly build prototypes. If such access were allowed, it was likely that too many people would access the same module at the same time, causing skew. This problem was solved by the use of the VAX DEC/CMS (Code Management System) software.

The specifications were made available at the same time as a prototype version, which acted as a "living" specification. The prototype included some of the major interface changes, such as integration with DEC/CMS. Users were able to try out the proposed interfaces and comment on what they liked or did not like. In some cases, the developers made available multiple solutions that provided the same underlying functionality to determine which possible solution suited the users best. This approach enabled the team to make decisions on functionality based on users' feedback early in the development process. The developers avoided the much longer process of first developing one solution, waiting for feedback, issuing a modified solution, and continuing the steps until results are attained.

Complicating the prototype effort was the need to continue to maintain the released version 1.1. DEC/CMS met the developers' need to control multiple simultaneous development threads. DEC/CMS maintains "elements," which are all of the versions of a single file stored as deltas from the original version. Each particular version of the element file is called a "generation."

A generation from which another generation is derived is called an "ancestor" generation. The trail from a generation through all of its ancestors back to the first generation of the element is called a "line of descent." All elements have a main line of descent, which consists of generation 1, followed by generation 2, and so on. In addition, DEC/CMS allows lines of descent to branch off from the main line into what are called "variant" lines. These variant lines of descent exist in parallel to each other and to the main line. Changes made to one variant, for example, are not reflected in the other lines of descent unless the changes are explicitly merged across lines.

Before any prototype work was begun, the DEC/Test Manager team agreed that all nonproduction code would be replaced into the group code library as variants, leaving the main line for maintenance and production version 2.0 development.

CMS classes — sets containing one specific generation of each element in the set — were used to represent complete prototypes, such as CMS_INTEGRATION and RESUBMIT. A developer working on a particular prototype could then retrieve modules by telling CMS that he wanted the latest generation on the same line of descent as the generation in the particular class:

```
CMS FETCH module.bli
    /GENERATION=cms_integration+ " "
```

These classes were also used with DEC/MMS (Module Management System) software to perform builds. DEC/MMS works from a description of objects to be built and their dependencies on other objects. The tool searches CMS libraries for components and makes use of CMS classes if told to do so.

All individual prototypes were also included in a more global class, PROTOTYPE, to allow the team to construct a single, executable prototype version containing all current prototype efforts. In some cases, the same module had been modified differently for several of the individual prototypes, and these modifications had to be reconciled for the combined prototype. DEC/CMS RESERVE/MERGE functionality was of major assistance. This procedure combines two generations from different lines of descent, automatically including independent changes and flagging different changes to the same region of the file (called merge "conflicts") for human resolution.

Once CMS performed its merge and the merge conflicts were resolved, the team used CMS's ability to compare a file against generations stored in a CMS library. This comparison was made to ensure the merged file contained code that made sense and that was expected to be there. This step was crucial to avoid accidental loss of code during the conflict-resolution process and to ensure that the automatically merged code sections were still valid BLISS.

A single executable version was built from the merged prototypes and made available throughout Digital on the DECnet network. Notification of the availability was posted in NOTES and was sent by VAX Mail directly to all known installed sites.

A single executable image was used because the team felt that a user was more likely to try the various prototypes if only one image had to be used.

Feedback on the prototypes was gathered from NOTES and Mail. The team directly polled installed sites, asking about the specific questions the prototypes had been designed to answer.

During the design phase, thousands of CMS merges were completed without problem. Without this merge capability, it would have been impossible to allow the developers access to all modules at the same time. Without concurrent access to all modules, the developers could not have built as many prototypes as quickly as they did. Further, without these early prototypes, the participation of internal users would not have been as high, and version 2.0 then would not have been built in the required timeframe.

DEC/Test Manager and Performance and Coverage Analyzer Integration

Testing was begun early in the development cycle to find and fix problems as early as possible. The existing DEC/Test Manager test set was run against the prototype image to check for regressions in nonprototype areas of the code. Some of the more comprehensive prototypes touched many modules; therefore, it was important to ensure preexisting functionality remained unchanged. In addition, some new tests were written for the prototypes, the purpose of which was not to test the prototype for correctness. Rather these tests could be run with the VAX Performance and Coverage Analyzer (PCA) to evaluate the relative performance of different prototypes and also to help tune the design.

The DEC/Test Manager and the PCA can be used together in an integrated fashion. This integration allows programmers to use good-coverage regression test suites as performance tests, merely by changing PCA collection from coverage information to performance statistics. For the DEC/Test Manager version 2.0 development, each developer was required not only to create and run the regression test suite but also to do code-path coverage analysis on his code before it was checked back into the master source library. The team was striving to have 90 percent of the code paths covered, and PCA allowed us to check how we were doing throughout development. The same tests were used to gather data on the poorly performing commands so developers could identify which routines/modules needed to be looked into.

While analyzing the prototype performance, the developers ran PCA over the rest of the code to identify the sources of several known performance problems in version 1.1. The results from this performance analysis led to the redesign of several key modules. The redesign produced no user-visible functional changes but significantly improved the performance of the commands most often used.

Implementation

Once the feedback from the prototypes and design reviews was incorporated into the designs and the specification and design phase was closed, implementation of DEC/Test Manager version 2.0 was begun.

The effort was approached in three different ways. First, some of the prototype code was good enough to be kept, with only a little time spent to make it production quality in the handling of error cases. Then, the variant generations in the CMS library were merged back onto the main line of descent — now the version 2.0 development stream.

Second, the developers took as much relevant existing code as possible from other Digital projects. This existing code was modified and reused in the DEC/Test Manager version 2.0. Modification of the existing code for the Test Manager environment proceeded far more quickly than writing new code. Included among the code the team reused was the pseudoterminal driver from an internal tool called PTYCON-32. Also adopted and upgraded to handle VT200-series terminals was the EDT group's terminal simulator for building in-memory screen pictures for VT52 and VT100 terminals.

Third, new code was written for all remaining functionality. No new code was considered complete until a code review was held. The DEC/Test Manager team then adhered to the software engineering principle of frequent, small integrations rather than one large integration. Therefore, every evening during active development, a procedure was run first to determine if code was checked back into the master CMS source library, and second to start a system build if it was warranted. As a result of this procedure, all recent code was always available to developers as they were writing new modules. Problems were found early rather than at a later, larger integration at the end of a base level.

Also, as part of the build procedure, an automatic test set execution was done. Just as the incremental integrations had done, these test sets kept the team aware of problems with the system. Moreover, the test sets were always available so implementors could easily schedule time for test review.

We were able to detect and fix bugs early, rather than have bugs mushroom into larger problems as more code was added. Problems were usually identified and solved while the code was still fresh in the developers' minds; therefore, time for bug fixes was reduced.

The use of the DEC/Test Manager to test itself during development was also beneficial as an early problem-detection system. In addition, a number of early versions of the software were released to internal Digital users. These users helped to identify problems early, during the implementation phase, when problems are often easiest and least expensive to fix and have the least impact on the project schedule. Because these early versions were used and refined for weeks internally, serious problems never reached the customer field test sites. For example, internal user feedback indicated that some changes would be required in the new functionality. This valuable feedback caused a complete interface change; however, the work was completed before external field test was begun.

During implementation, NOTES and a QAR system were used for problem reporting and tracking. This reporting enabled the developers to associate problems with sections of code, to report progress, and to associate it with a test in the DEC/Test Manager library. A project rule required that all bug fixes have a test in the library and a PCA coverage analysis performed before the bug was considered fixed. As a result of enforcing this procedure, the DEC/Test Manager team achieved a code coverage of 87 percent, finding two thirds of the problems themselves.

Conclusion

The DEC/Test Manager team was able to produce 30,000 lines of prototype code (thrown away) and 60,000 lines of tested, production-quality code in under seven months as a result of several key factors: creative use of project procedures and tools, team commitment to these procedures, and the use of the developing product by the team as well as many internal users. It has been two years since DEC/Test Manager version 2.0 became available to customers, and fewer than six unique problems have been reported by the customer base.

References

1. B. Beander, "VAX/VMS Software Development Environment," *Digital Technical Journal* (February 1988, this issue): 10–19.

2. A. Duncan and T. Harris, "Software Productivity Measurements," *Digital Technical Journal* (February 1988, this issue): 20–27.

3. P. Gilbert, "Development of the VAX NOTES System," *Digital Technical Journal* (February 1988, this issue): 117–124.

Peter D Gilbert

Development of the VAX NOTES System

The VAX NOTES computer conferencing system is a communications tool for on-line discussions. This paper discusses the innovative strategies devised by the VAX NOTES team for the development of this system, including the decisions to build a prototype, to perform human-factors engineering, and to include a technical writer early in the development cycle. Also described in this paper are several key product features, with emphasis on their effect on product performance and extensibility: the multitasking, multithreaded server; the user interface; the underlying storage medium; and the callable interface.

Computer conferencing is a software tool for ongoing discussions among individuals. Users can asynchronously read and write messages in the conference at times suitable to themselves. The computer conference provides an organized structure and a permanent record of users' messages, or discussions. Computer conferencing is a viable substitute for conventional meetings, offering conspicuous savings in space, time, and money.

The VAX NOTES system is a computer conferencing tool designed for use on the VAX/VMS operating system. The development effort was successful, meeting its requirements and schedule and incorporating several innovations. The VAX NOTES system is used by Digital's hardware and software engineers for structured project communication and has found similar favor with customers.

The success of the VAX NOTES system is largely due to the design and development strategies used in its creation, and to the context in which it was developed. This paper discusses the rationale behind these decisions and their effect on the product. This retrospective may be useful to other software developers.

First, we briefly discuss the origins of computer conferencing at Digital.

A Brief History of NOTES at Digital

Early in 1980, a Digital engineer wrote a computer conferencing program, K-NOTES, that had its roots in the PLATO system developed at the University of Illinois. The K-NOTES program was written as a test of the newly added indexed file support in the VAX PL/I language. VMS software engineers also used K-NOTES to log VMS design changes that might be of interest to others. K-NOTES was a crude but effective tool for communication and became popular with other engineering groups within Digital. Indeed, it was often mistaken for part of the VMS operating system.

Software developers using PDP-11 systems also wanted access to computer conferences. A software developer wrote the NOTES-11 program as a "midnight project" — an unfunded project developed outside normal working hours. Originally developed for the RSTS/E operating system, NOTES-11 was later adapted to run on the RSX, VMS, and P/OS operating systems. Many of the enhancements suggested for K-NOTES were incorporated into NOTES-11, including support for multiple time zones and a per-user record of conferences (later called the "notebook").

As Digital's engineering network grew to hundreds of computers connected by DECnet software, the number of computer conferences abounded. Most products had a conference for Digital users to ask questions, report possible problems, and suggest enhancements. There were also conferences on a diversity of other subjects of general and personal interest, such as security, smoking policy, and jobs.

VAX NOTES Project Decisions

In late 1984, VAX NOTES became a funded project with two developers and a one-year development schedule.

We looked at several competitive conferencing systems. Nearly all of these systems were oriented toward hardcopy terminals. Most included an integrated electronic mail system or personal messaging capability, and most had good basic documentation. We found that in many of these systems, navigation through the messages in a conference was difficult.

None of the systems provided good support over a computer network, none integrated well with the underlying system, and none offered a callable set of routines for accessing conferences.

Based on our study, we decided our product should offer users the following:

- A screen-oriented terminal interface
- An interface to the VMS Mail utility
- An easily conceptualized structure with simple navigation (A comb-like structure of topics and replies was chosen.)
- A distributed conferencing system that makes efficient use of DECnet capabilities[1,2]
- Simple, introductory documentation for nontechnical and novice users

A server-based technology layered on DECnet software was clearly needed to efficiently support distributed conferencing.

During the planning of the project, we also determined that the product should be extensible, which is discussed in the section Extensible Design through the Callable Interface. In addition we decided to use the VAXTPU (Text Processing Utility) software to implement the user interface, and VAX RMS (Record Management Services) software for the underlying storage mechanism. Discussion of the use of a server, the user interface, the storage features, and the callable interface can be found in the section Design of Main Features.

These product decisions were coupled with decisions about how to structure the development process. We decided to build a prototype. Also, we worked with the Software Usability Engineering (SUE) Group to perform human-factors testing.[3] Finally, we included a technical writer early in the development cycle. These decisions are discussed in the next section.

Our development tools included VAX DEC/CMS (Code Management System) software for source control and VAX DEC/MMS (Module Management System) software for doing incremental builds. In addition, the NOTES utility itself became an important development tool.

Because of limited resources and time, we made no systematic test suite. We correctly assumed that an internal field test with thousands of users would provide an incredible amount of testing. We discuss our test results in the section Field Test at the end of this paper.

Development Decisions

This section describes three elements of the development strategy used for the VAX NOTES project. Each of these — the prototype, usability testing, and early inclusion of a technical writer — is a somewhat novel approach and led to the project's success.

Reasons for Building a Prototype

Several of our product decisions conspired to make a prototype very desirable. First, our untried uses of the new VAXTPU utility imposed some risks. Second, we had decided to perform human-factors testing; our short development cycle required that some form of the product be available early for this testing. Finally, a prototype would give the developers more experience with server-based applications and with TPU. In short, we expected to learn from the prototype. (In addition to meeting our project needs, the prototype could be used within Digital to provide better access to remote conferences while the VAX NOTES system was being developed.)

We only required that the prototype would support core features. The prototype would use the same conference storage format as was used by NOTES-11. Therefore, features that required changes in the storage format (such as keywords and membership lists) were necessarily absent from the prototype. Also, some rarely used operations were left unimplemented in the prototype since NOTES-11 could be used instead. Indeed, the prototype was incapable of even creating a new conference.

We decided to include in the prototype one additional feature called the notebook. The notebook is a per-user record of the conferences in which the user participates and of the notes that have been read in those conferences. From our experience with the notebook feature in NOTES-11, we anticipated difficulties in providing a smooth integration between the notebook and the rest of the VAX NOTES interface. The notebook feature was included in the prototype so that human-factors tests could help us resolve any problems.

Usability Engineering of NOTES

We enlisted Digital's SUE Group to help evaluate and improve the VAX NOTES system's ease of use.

The first task was to create a measurable definition of usability. This definition allowed us to measure and improve the VAX NOTES user interface. The SUE group helped us identify our goals and offered guidance to help us reach them. The usability engineering approach to software development is described in the paper "Software Usability Engineering," this issue.[3]

The VAX NOTES usability definition included measurements for 10 attributes. These are

- Initial use
- Learning rate
- Infrequent use
- Compatibility with NOTES-11
- Compatibility with other competitive conferencing systems
- Initial evaluation
- Casual evaluation
- Mastery evaluation
- Error recovery
- Fear of feeling foolish

The first three attributes were measured by user performance on NOTES benchmark tasks. For these, the metrics were the number of errors and the number of successful interactions made by the users in 30 minutes. The compatibility and evaluation attributes were measured with attitude questionnaires containing a semantic differential. The final two attributes were measured with unique questionnaires.

The testing centered on the initial use, learning rate, and initial evaluation attributes.

For initial performance, the goal was for initial users to have three or four successful interactions on a benchmark task in their first 30 minutes of using VAX NOTES; 8 to 10 successes was considered the best case. The learning rate was measured by comparing the performance on a second benchmark with the first, where the performance was measured as a work speed relative to a practiced expert performing the same tasks. The goal for initial satisfaction was fairly positive, 1.0 on a scale from -3.0 to 3.0.

In the first VAX NOTES tests, users exceeded our best-case level for initial use, and so we adjusted our goal. The learning rate was acceptable, but the evaluation score fell just short of our goal. We then made changes to the interface as a result of the SUE Group's impact analysis. With these changes, VAX NOTES eventually met or exceeded our usability goals.

Inclusion of a Technical Writer

We were able to include a technical writer on the development team shortly after the prototype was completed, rather than after implementation as is often the case. The writer's review and exposition of the planned interface was concurrent with (and sometimes preceded) its implementation. The writer's most important role and one of our goals was to ensure that the system could be clearly documented for nontechnical users.

As a member of the development team, the writer made technical changes to simplify features that were difficult to explain, made the set of commands more self-consistent, and suggested other improvements to the implementors.

Because the technical writer was involved in the process early, problems with either the code or the documentation could be resolved early, and a combination of both code and documentation changes could be used in the solution. The early involvement of a technical writer also gave us high-quality documentation (and on-line help) for field test and some of the human-factors studies, thereby allowing the whole system to be evaluated.

Design of Main Features

In this section we discuss the reasons for including a server, creating the user interface with TPU, selecting the VAX RMS software as a storage medium, and designing a callable interface that is extensible. We also discuss how these features were implemented and what trade-offs were made.

Use of a Server

The K-NOTES and NOTES-11 tools relied on RMS and the DECnet software to transparently access conferences on remote systems. Although the VAX RMS software provides efficient access to individual records in a remote file, most NOTES operations require multiple RMS operations. For example, to list a directory of the notes in a conference would require one RMS operation per note. This performance is efficient when the conference is stored on the same system that the user is on. However, when the conference is several slow network links away from the user, the round-trip delays for every RMS record operation are seen as frustrating delays at the user interface. The advantages of using a NOTES server running on the system that hosts the conference were clear. The server would

- Perform many local RMS operations without a network delay
- Support sophisticated requests more efficiently than RMS
- Send back only the information that the user requested
- Allow DECnet software to send larger, hence, fewer network packets with fewer transmissions, resulting in a tenfold improvement in the time taken to satisfy users' requests

A multitasking server can handle requests from one or more users. Because more requests can be handled, fewer server processes, hence fewer resources, are needed on the system hosting conferences. Also, VAX RMS software pools buffers, which offers another advantage when several users are reading the same conference: a user's request may be satisfied from a buffer that had been read for a different user's earlier request.

Multitasking gives the server process something to do between an individual user's requests. For example, while a user is reading a note, the server can process other requests. The multitasking usually does not have much effect on the response time seen by the users.

The prototype server employed multitasking. The VAX NOTES server used both multitasking and multithreading.

When the multitasking server receives a request, it builds a control block to contain the request and queues it onto a list. The synchronous code is in a loop that processes the requests; the code removes a control block from the queue, performs the operation, and sends back the results. If there are more results than can be returned in a single packet, a continuation control block is enqueued, and later activated by receipt of a "send me more" request from the user.

The prototype processed each request either to completion or until the results filled the server's response buffer. A disadvantage with this approach is that some requests may take a considerable amount of time before returning any results. For example, a request to search for a string within a note could cause significant delays for other users of the server. This problem was solved by making the server multithreaded.

The multithreaded VAX NOTES server can switch from one task to another even before the first task has completed. The implementation proved to be far easier than we had anticipated, and the multithreaded server simplified the handling of users' "send me more" requests. Each "thread of execution" has a separate stack and set of registers. Whenever the server does an I/O operation, it does so without waiting and tells the "scheduler" to find and work on another request.

When the I/O completes, the corresponding request block is again enqueued for processing. Because I/O is invariably done for a request, there is no need for a sophisticated scheduler. The "scheduler" saves the registers on that thread's stack, searches for a thread that can proceed, restores its registers, and continues the execution of that new thread. Since the VAX NOTES system does relatively little processing between I/O operations, no user is able to significantly affect the response time seen by other users.

The VAXTPU Programmable Editor

VAXTPU software is a programming language and interpreter that is shipped with the VMS operating system. TPU is especially suited for writing text editors and has been used to write the human-engineered EVE editor, the ACL (Access Control List) editor, and VAX LSE (Language-Sensitive Editor).

Although the VAX NOTES system is not an editor, we wanted an interactive screen-oriented user interface. We expected that VAXTPU would provide a good language in which to write the interface, since the functions for handling the screen, key definitions, strings, and text manipulation are built in to the TPU language. The VAXTPU utility also provides a callable interface, so that it can call and be called by programs written in other languages. By using these features of the VAXTPU language, we could economically produce a desirable user interface.

However, TPU's callable nature had not yet been used in anything as complicated as the VAX NOTES product, and we were unsure whether its performance as an interpretive language would adversely affect the perceived performance. The prototype helped resolve this issue early, and the performance proved to be far better than we needed.

The decision to use TPU had a considerable and unexpected payoff: because of its interactive and interpretive nature, it was possible to greatly reduce the time spent in the compile-link-run cycle. After the initial investment in getting NOTES and TPU "talking" together, typical development of the TPU code followed this procedure:

- Run the VAX NOTES utility.
- Read the TPU code into a text buffer (by a command issued at the NOTES prompt).
- Edit the code.
- Select and recompile the modified procedures in the code. This was done with only a few keystrokes. Developers had defined a key to mean "compile the selected code."
- Exit the buffer; return to the NOTES prompt.
- Test the change.

This procedure continued until the developer was happy with the changes. Then the buffer containing the TPU code would be written out to a file, and the developer could either exit the NOTES utility or proceed with other changes. The power of this approach resulted in an estimated development-time savings of four person-months.

We discovered, however, a disadvantage with this evolutionary approach. The resulting TPU code tended to be poorly commented because the code would be working before there was any need for comments. We solved this problem with an occasional "revolution": we reviewed the code as a whole, improved the organization, made it more systematic, and added the comments necessary to understand and maintain the code.

The use of the VAXTPU software was not without pitfalls. The VAX NOTES team made the most sophisticated use of TPU software to date and discovered several interesting bugs in the TPU software. Because the development groups were in close proximity to each other, little time was lost waiting for fixes or workarounds. But not all of the problems were resolved; for example, TPU's control-C handling was designed for simple editors, not complex applications, and control-C handling remained a weak point in the first release of VAX NOTES.

VAX RMS Software for Storage

We had several good reasons for choosing the VAX RMS software for our underlying storage medium.

- VAX RMS software is bundled with the VMS operating system.
- The code is robust.
- VAX RMS performs well on large indexed files. (We wanted an entire conference to be held within a single file.)
- RMS supports concurrent, coordinated access and updating of records.
- We were familiar with the VAX RMS software, therefore we would not need to spend time gaining experience with it.

Although these reasons were compelling, we considered some alternatives because of the following issues. The NOTES software's view of the information in a conference would be more akin to a true database than to an indexed file. Therefore, the use of indexed files would require that development time be spent implementing a makeshift database atop indexed files. Also, the VAX NOTES system would be enhanced if it and other tools used a relational or object database.

Despite these considerations, we decided in favor of RMS based on the benefits of using RMS listed above, our interest in not adding licensing costs for customers, and a tight schedule.

We did, however, decide to insulate the higher functions in NOTES from RMS with a callable interface. This permits a future release of the VAX NOTES system to use a different underlying storage medium without affecting code that uses the callable interface.

The VAX NOTES system stores data in the RMS records with a type-length-value (TLV) encoding called Digital Data Interchange Syntax (DDIS). Each datum in a record identifies itself by a "type" code and is followed by the length and actual value of the datum. For example, the "header" information in each note in a conference may include the author's name, the date the note was written, a title, and a list of keywords. A sample note header with TLV encoding follows:

```
{author} 5 "BYRNE"
{date} 8 "22-JAN-1979 10:30:27"
{title} 15 "Crystallography"
```

The {author}, {date}, and {title} are suitably encoded, and 8 bytes are used to represent the date (as on VMS).

The advantages of TLV encoding are that it is succinct and evolutionary. Additional information (such as coauthors or an edit history) can be added without changing the format or invalidating any existing data. Changes are made by simply creating a new type code for the new information, making minor changes within the callable routines, and providing the new information to the callable interface.

Extensible Design through the Callable Interface

Computer-conferencing has a practically unlimited range of possible uses and useful extensions.[4] We could not foresee all possible future enhancements, or provide even a tenth of the many requested features because of our development schedule. Resigning ourselves to the fact that we could not know what our code would eventually become, we resolved to provide extensibility that could support a variety of future directions.

One of the design decisions allowing for this extensibility is described in this section.

The design of the callable interface was a significant aspect of the VAX NOTES system development. This interface provides all the operations that access conferences and notebooks. The callable interface must meet several goals; it must

- Allow the underlying storage medium to be changed in a future implementation; for example, from RMS files to object databases
- Allow other user interfaces to be added without publishing (and compromising) the underlying storage format
  (Two interfaces used within Digital show the value of this approach. One is designed for offline — batch — reading and writing of notes; the other is an interactive interface implemented with another programmable editor — EMACS.)
- Have the flexibility to be extended to handle a variety of possible (and now unforeseen) future enhancements
- Provide a well-defined interface between the two halves of VAX NOTES: the user interface and the file access routines
- Map easily to remote procedure calls or network packet transfers
- Be reentrant, that is, the "context" of an operation must be specified in each call, and operations done with different contexts should be independent of each other (This independence also proved to be an advantage for implementation of the multithreaded, multitasking NOTES server.)
- Be capable of becoming a supported interface for use by customers

The callable interface proved quite successful in meeting these goals.

All the routines in the callable interface have the same format:

```
status = NOTES$object_operation
     (context, inputs, outputs)
```

The `object` is either NOTEFILE, CLASS, ENTRY, KEYWORD, NOTE, PROFILE, or USER. The `operations` BEGIN, END, ADD, DELETE, GET, and MODIFY are common to most of these objects. A few additional routines handle lists contained within an object, for example, NOTES$NOTE_GET_TEXT gets the next line of text from the note most recently accessed (with the specified `context` by NOTES$NOTE_GET.

The `context` parameter is an uninterpreted value defined by the NOTES facility. On a NOTES$object_BEGIN call, NOTES stores a nonzero value in the `context` parameter. That `context` is passed to other routines for access to that kind of object. A call to NOTES$object_END frees the resources used to maintain the context and zeroes the value.

The `inputs` and `outputs` are item lists. An item list is a list of one or more item descriptors, each of which specifies an item code. The item list is terminated by an item code of 0. Figure 1 depicts the structure of a single item descriptor.

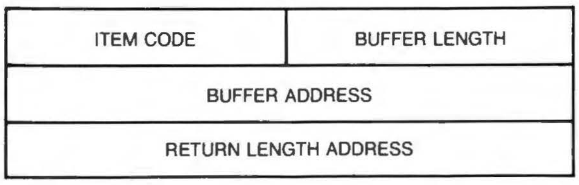


Figure 1 Structure of an Item Descriptor

The item code specifies the item of information that the caller is specifying (through the `inputs` parameter) or requesting (via the `outputs` parameter). The buffer length and buffer address describe the buffer (for inputs or outputs), and the return length address is the address of a location into which NOTES will write the actual length of the requested information (for outputs).

To see how this works, we consider the calls needed to read notes 1 through 5 from a local conference. The name of a file is specified as an input to NOTES$NOTEFILE_BEGIN, which opens the file and initializes the notefile context. The notefile context is passed as an input to NOTES$NOTE_BEGIN, and this establishes the note context. A call to NOTES$NOTE_GET requests notes 1 through 5. (The string "1–5" is passed as an input.) Then NOTES$NOTE_GET_TEXT is repeatedly called until a "no more text" status is returned. The process repeats: another call to NOTES$NOTE_GET is made (specifying a "continue" item code). Eventually NOTES$NOTE_GET returns the status "no more notes."

At any time, the same note context could be used to read other notes by omitting the `continue` item code. This would cancel reading of the current stream of notes (1 through 5, in the example) before handling the new request. This cancellation can be avoided by creating another note context, with another call to NOTES$NOTE_BEGIN. The resources used to maintain the note context are freed by a call to NOTES$NOTE_END, and similarly with NOTES$NOTEFILE_END, which deallocates memory and closes the file.

There is no separate call for opening a notebook. Although conferences and notebooks contain different kinds of information (in our interface), the storage formats are actually the same. Because the formats are the same, the variety in the code is reduced, and the inner routines can be reused (and stressed) in several different ways, making them more robust. This similarity offers another advantage. A future release could easily allow personal notes or annotations to be stored in the user's notebook, through the use of the existing NOTES$NOTE_operation routines.

Access to remote conferences is easily effected. An operation code (for the routine), the context, the inputs, and the requested outputs are "linearized" into a TLV format (DDIS) and sent to the NOTES server on the remote system. The routine is called, and the returned status and outputs are "linearized" and sent back. This can be viewed as a set of specialized Remote Procedure Calls (RPC).

The inputs to NOTES$NOTE_GET may also include "hints" to indicate, for example, whether the text of the note is desired. (The user may want to read the text or may simply want a directory of the notes.) If the operation is one that may be repetitive (such as NOTES$NOTE_GET), the server makes multiple calls to that routine so that it can buffer and send back larger packets of information. There are some complications in the handling of signaled exceptions and buffering, and in how the server validates the context, but they had little effect on the overall design.

Field Test

The internal field test was impressive. Within an hour of making the VAX NOTES system available within Digital, it was installed on four continents. Besides providing a popular tool on hundreds of systems on Digital's engineering network, we also provided the medium — a VAX NOTES conference — by which users could easily report problems and make suggestions. We were deluged.

These reports directly contributed to the quality and success of the VAX NOTES system.

Acknowledgments

Thanks are due the editors and reviewers of this paper, and the many people whose interest in computer conferencing influenced the VAX NOTES system, especially Mark Goodrich, Len Kawell, Valerie Rogers, Benn Schreiber, and Tom Spine.

References

1. *DECnet Digital Network Architecture (Phase IV) General Description* (Bedford: Digital Equipment Corporation, Order No. AA-149A-TC, 1982).

2. P. Beck and J. Krycka, "The DECnet-VAX Product — An Integrated Approach to Networking," *Digital Technical Journal* (September 1986): 88–99.

3. M. Good, "Software Usability Engineering," *Digital Technical Journal* (February 1988, this issue): 125–133 .

4. S. Hiltz, *The Network Nation: Human Communication via Computer* (Reading: Addison-Wesley, 1978).

Michael D. Good

Software Usability Engineering

Usability is an increasingly important competitive issue in the software industry. Software usability engineering is a structured approach to building software systems that meet the needs of users in various environments with varying levels of computer experience. This approach emphasizes observation of people using software systems to learn what people want and need from software systems. The three principal activities of software usability engineering are on-site observations of and interviews with system users, usability specification development, and evolutionary delivery of the system. These activities are parallel steps in the development cycle.

Computer system designers have not always adopted a user-centered perspective on software design. Instead, many designers resolved design questions about the human-computer interface by using introspective criteria such as personal preference or conceptual appeal.

This introspective approach to user-interface design might produce a usable system when software engineers represent actual users. However, computer systems today are being built for a wide range of people whose needs often have little in common with the needs of system designers.

In response to market demand for systems that satisfy a growing and varied user community, usability is becoming an increasingly important competitive issue. Designers are striving to create computer systems that people can use easily, quickly, and enjoyably. Indicative of this trend is increased membership since 1982 in professional groups such as the Association for Computing Machinery's Special Interest Group on Computer-Human Interaction (ACM SIGCHI) and the Computer Systems Group of the Human Factors Society.

Digital's Software Usability Engineering Group believes that engineers must learn about the needs and preferences of actual users and should build systems to accommodate them. With an understanding of customer environments, an awareness of technological possibilities, and imagination, we have produced many ideas for products that meet users' needs.

The Software Usability Engineering Process

The role of engineering is to apply scientific knowledge to produce working systems that are economically devised and fulfill specific needs. Our software usability group has adapted engineering techniques to the design of user interfaces. To understand user needs, engineers must observe people while they are actually using computer systems and collect data from them on system usability. Observation and data collection can be approached in the following ways:

- Visiting people while they use computers in the workplace
- Inviting people to test prototypes or participate in usability evaluations at the engineering site
- Soliciting feedback on early versions of systems under development
- Providing users with instrumented systems that record usage statistics

Our group uses these methods to gather information directly from users, not through secondhand reports. We use these methods to study the usability of current versions of our products, competitive systems, prototypes of new systems, and manual paper-based systems.

Our software usability engineering process evolves as we use it in product development. As of 1987, the process consists of three principal activities:

- Visiting customers to understand their needs. By understanding a customer's current experience with a system, we gain insight into our opportunities to engineer new and better systems. We collect data on users' experiences primarily through contextual interviews, that is, interviews conducted while users perform their work.
- Developing an operational usability specification for the system. We base the system specification on our understanding of users' needs, competitive analysis, and the resources needed to produce the system. This specification is a measurable definition of usability that is shared by all members of the project team.
- Adopting an evolutionary delivery approach to system development. Developers start by building a small subset of the system and then "growing" the system throughout the development process. We continue to study users as the system evolves. Evolutionary delivery is an effective method for coping with changing requirements — a fundamental aspect of the development process.

These three development activities are parallel, not sequential. We do not view user-interface design as a separate and initial part of the development process but as an ongoing process in system development.

These usability engineering techniques apply to most software development environments and are most effective in improving software usability when applied together. However, designers who use any single technique can improve a system's usability. Our group has used this process in the development of several of Digital's software products, including the EVE text editor and VAXTPU (Text Processing Utility) software, VAX NOTES software, MicroVMS workstation, VAX Software Project Manager, VAX COBOL Generator software, VAX Language-Sensitive Editor, and VAX DEC/CMS (Code Management System) software.

Visiting Customers to Understand Their Needs

Data collected at the user's workplace provides insight into what users need in both new and modified systems. During interviews of users actually working with their systems, we ask about their work, about the details of their system interfaces, and about their perception of various aspects of the system. The user and the engineer work together to reveal how the user experiences the system as it is being used. These visits with users are the best way for engineers to learn about users' experiences with the system.

Ideally, the number of interviews conducted per product depends on how much data is being generated in each succeeding interview. The interview process stops when new interviews no longer reveal much new usability data. In practice, resource and time limitations may stop the interview process before this point. In any event, our approach is to start with a small number of interviews (four or less) with people in various jobs. We use these interviews to determine how many and what type of users will be most useful for uncovering new usability data.

Information Gained in Field Studies

Contextual interviews reveal users' ongoing experience of a system. Other types of interviews, which are not conducted while the user works, reveal users' summary experience, that is, experience as perceived after the fact. Data on ongoing experience provides a richer source of ideas for interface design than data on summary experience.

For example, data collected from field studies has revealed the importance of interface transparency to users. A transparent interface allows the user to focus on the task rather than on the use of the interface. Our understanding of transparency as a fundamental usability concept comes from an analysis of data on ongoing experience.

Some interface techniques can help keep the user in the flow of work, thus increasing interface transparency. One example can be drawn from a workstation application for desktop publishing. Pop-up menus that appear at the current pointer location create a flow of interaction that reduces mouse movement and minimizes disruption to the user's task. Users do not have to move their eyes and hands to a static menu area to issue commands, making this an effective interface feature for experienced users.

We will consider using pop-up menus in new workstation software applications when we believe their use will keep the user in the flow of work.

We have developed our understanding of transparency by observing people using a variety of applications in different jobs. Transparency is an aspect of usability that we find across many different contexts. In developing new products, it is also important to consider the diversity of environments in which people will use the system. Different users in different contexts have different usability needs. Some important aspects of user's context are

- Type of work being performed
- Physical workplace environment
- Interaction with other software systems
- Social situation
- Organizational culture

All these aspects influence the usability of a system for each individual. As with other products, software systems are used in the field in ways not anticipated by the designers.

Because the context in which a system is used is so important, we interview a variety of users who use particular products to perform different tasks. We look for common elements of usability for groups of people, as well as the distinctive elements of usability for individual users.

Conducting Contextual Interviews

Interviewers bring a focus, or background,[1] to their visits with users. The focus determines what is revealed and what remains hidden during a visit. The engineer needs to enter an interview with a focus appropriate to achieve his goals. For example, in some visits an engineer may need to look for new product ideas; in others, the engineer may need ideas to improve an existing product.

To avoid losing data, interviewers should not try to extensively analyze their data during the session. We use two-person teams, where one team member concentrates on the interview and the second member records the data. Contextual interviews rapidly generate large amounts of data. The data derives from an understanding of a user's experience of a system, as shared by a user and an interviewer. To generate such data, interviewers need to concentrate on their relationships with users and understand what users do during the session.

Whenever possible, we videotape interviews. If users are unwilling to have their work videotaped, we audiotape the session while the second team member takes detailed notes to supplement the taped information. The two team members meet after the interview to reconstruct an accurate record of events.

Even without any taping or note-taking, engineers can learn a great deal from user visits. Although the detail from the interview may not be remembered, the understanding gained during the interview is still a valuable source of insight.

Developing an Operational Usability Specification

Studying users provides a rich, holistic understanding of how people experience software systems. However, each person will have his or her own interpretation of user experience as it relates to usability. Similarly, a team of people working on a project will find that each member has a different understanding of what "usability" means for that product. Keeping these understandings private and unarticulated can have two undesirable results. First, team members work toward different and sometimes mutually exclusive goals. Second, the team does not have a shared criterion for what it means to succeed or fail in meeting users' needs.[2]

Our group constructs shared, measurable definitions of usability in the form of operational usability specifications. These specifications are an extension of Deming's idea of operational definitions.[3] We based our usability specifications on the system attribute specifications described by Gilb[4] and Bennett.[5] A usability specification, described in the following section, includes a list of usability attributes crucial for product success. Each attribute is associated with a measuring method and a range of values that indicates success and failure.

Constructing a Usability Specification

The development of the VAX NOTES conferencing system provides an example of a usability specification.[6] Table 1 is a summary of the usability specification for the first version of the VAX NOTES system. Five items are defined for each attribute: the measuring technique, the metric, the worst-case level, the planned level, and the best-case level.

Table 1 Summary Usability Specification for VAX NOTES Version 1.0

Usability Attribute	Measuring Technique	Metric	Worst-Case Level	Planned Level	Best-Case Level
Initial use	NOTES benchmark task	Number of successful interactions in 30 minutes	1–2	3–4	8–10
Initial evaluation	Attitude questionnaire	Evaluation score (0 to 100)	50	67	83
Error recovery	Critical-incident analysis	Percent incidents "covered"	10%	50%	100%

The measuring technique defines the method used to measure the attribute. Details of the measuring technique (not shown in Table 1) accompany the brief description in the summary table. There are many different techniques for measuring usability attributes. We have usually measured usability attributes by asking users to perform a standardized task in a laboratory setting. We can then use this task as a benchmark for comparing usability attribute levels of different systems.

In the VAX NOTES case, we chose to measure initial use with a 14-item benchmark task that an expert VAX NOTES user could finish in three minutes. Initial users were Digital employees who had experience with the VMS operating system and the Digital Command Language but not with conferencing systems. The users completed their initial evaluations using 10-item Likert-style questionnaires after they finished the benchmark task. Error recovery was measured by a critical-incident analysis. In the analysis, we used questionnaires and interviews to collect information about costly errors (critical incidents) made by users of the prototype versions of the VAX NOTES software.

The metric specifies how an attribute is expressed as a measurable quantity. Table 1 shows the definitions of the metrics in the VAX NOTES specification. For the initial-use attribute, the metric was the number of successful interactions in the first 30 minutes of the benchmark task. For the initial-evaluation attribute, we scored the questionnaire on a scale ranging from 0 (strongly negative) to 100 (strongly positive), with 50 representing a neutral evaluation.

For error recovery, the metric was the percentage of incidents reported with the prototype systems that would be "covered" (i.e., eliminated) by changes made in version 1.0 of the VAX NOTES system.

The worst-case and planned levels define a range from *failure to meet minimum acceptable requirements* to *meeting the specification in full*. This range is an extension of Deming's single criterion value, which determines success or failure. It is easier to specify a range of values than a single value for success and failure. Providing a range of values for several attributes also makes it easier to manage trade-offs in levels of quality of different attributes.

The best-case level provides useful management information by estimating the state-of-the-art level for an attribute. The best case is an estimate of the best that could be achieved with this attribute, given enough resources.

For the initial use of VAX NOTES software, we defined the planned level as experiencing 3 or 4 successful interactions in the first half hour of use. We considered 1 or 2 successful interactions to be the minimum acceptable level, and 8 to 10 successful interactions to be the best that could be expected. In practice the actual level was 13 successful interactions, suggesting that we set the levels for this attribute too conservatively.

The planned level for initial evaluation (67) was fairly positive. Users' neutral feelings were acceptable but negative feelings were not, so we set the worst case at 50. We set the best case at 83, which represented the highest scores we had seen so far when using this questionnaire with other products. The actual tested value was 67, matching the planned level.

We planned an error-recovery level that could cover 50 percent of the reported critical incidents. The worst-case level was set at a fairly low 10 percent, whereas the best case would be to cover all of the reported critical incidents. In practice, 72 percent of the critical incidents were covered, exceeding the planned level.

Many usability specifications provide further detail by including "now" levels and references. Now levels represent current levels for an attribute, either for the current version of the product or for competitive products. References can be used to add more detail, such as describing how the levels were chosen, and to document the usability specification.

User needs and expectations are shaped in part by the marketplace; therefore competitive analyses can provide important data for usability specifications. We have constructed usability specifications that compare the system under development to either the current market leader, the product with the most highly acclaimed user interface in the market, or both. We can also compare the systems by measuring usability on appropriate benchmark tasks.

Limitations of Usability Specifications

Constructing a usability specification helps build a shared understanding of usability among the diverse people working on a development project. However, to achieve a shared understanding, trade-offs have to be made. Usability specifications represent a constricted and incomplete definition of usability. The analytic definition of usability is necessarily less complete than an individual's holistic understanding based on observing people use systems.[7] Nonetheless, we deliberately trade off the holistic understanding for the analytic definition because the latter economically focuses our efforts on essential elements of product usability.

If engineers do not understand the needs of users before creating a specification, they risk developing a specification that does not reflect users' needs. As a result, the product that meets its specification might still be unusable or commercially unsuccessful. Development teams must continually evaluate usability specifications during the development process and make the changes necessary to reflect current information on users' needs. This approach is part of evolutionary delivery, described next.

Adopting Evolutionary Delivery

Changing requirements pose a challenge in user-interface design as they do elsewhere in software development. Brooks refers to changeability as one of the essential difficulties of software engineering — a problem that is part of the nature of software engineering and that will not go away.[8]

Evolutionary delivery exploits, rather than ignores, the changeable nature of software requirements.[4] This technique has been referred to as incremental development[8] and as iterative design.[9] We believe that "iterative design" is usually a redundant term in software design. Unless otherwise mandated by external sources, most software design is already an iterative process.[10] The waterfall model and similar models of software design are useful for managing project deliverables, but they do not describe what happens in software design and development. Evolutionary delivery takes for granted the iterative nature of the design process, rather than treating iteration as an aberration from textbook methods.

Evolutionary delivery is the process of delivering software in small, incremental stages. An initial prototype subset of the software is built and tested. New features are added and existing features refined with successive versions of the system. The prototype evolves into the finished product.

Evolutionary delivery helps to build the project team's shared understanding of the system's user-interface design. Contemporary direct-manipulation user interfaces are too rich, dynamic, and complex to be understood from paper specifications. Even simpler terminal-based interfaces are too involved to be understood completely without being seen in action. Early delivery of subset systems helps everyone on the development team understand the system being designed, making it easier to build a shared vision of the final system.

Early, incremental deliveries also demonstrate project progress in a concrete form. Demonstrating improvements to the system at the user-interface level can be an important factor in maintaining managerial support for a project and continuing availability of resources.

The techniques used to improve system usability during the stages of evolutionary delivery include the following:

- Building and testing early prototypes
- Collecting user feedback during early field test.

- Instrumenting a system to collect usage data
- Analyzing the impact of design solutions

These general-purpose techniques can be used independently of an overall usability engineering process. They are described in the following sections, some with examples from the evolutionary delivery of the EVE text editor.[9,11]

Building and Testing Prototypes

The first step in an evolutionary delivery process is building and testing prototypes. These prototypes effectively test for ease of learning[12] and can provide the germinal product. Prototyping also helps identify potential interface problems while still very early in the development cycle.

From the point of view of usability engineering, the first prototype subset produced should facilitate usability testing. This typically means that the system

- Includes only simple versions of the most important and most frequently used features of the product
- Is able to complete a simple benchmark task that the designer will use for a preliminary evaluation of the system's usability attributes
- Is useful only for limited testing, not for normal work

If the first prototype is actually useful for normal work, it is probably a larger portion of the project than needs to be delivered at this stage.

The first prototype of the EVE text editor was available three weeks after development began. This prototype tested only the keypad interface. At that point, we had neither implemented nor fully designed the command-line features. To test ease of learning, seven new computer users used EVE in informal laboratory sessions. They performed a standard text-editing task. The tests showed that the keypad interface was basically sound; only minor changes to the basic EVE keypad commands were required. This prototype was the first of 15 versions of EVE that users tested over 21 months.

Because prototypes are not suitable for daily use, they must be tested in controlled conditions. For example, the test might involve asking users to complete a standardized task, where that task is the only one that can be completed using the prototype system. Special equipment can make it easier to conduct these tests and to collect more complete data, but is not necessary. For example, videotaped records can help in later analyses, but as with user visits, we can learn much without them.

For many years we tested prototypes in spare offices, developers' offices, or users' offices. Our group now tests most prototypes in our usability engineering laboratory, which is equipped with computer hardware and software, a one-way mirror, and videotaping equipment. The laboratory resources provide greater opportunity for routine testing and elaborate data collection.

Collecting User Feedback during Early Field Test

The earlier a system can be delivered to a group of users for field test, the sooner valuable information will be available to designers. User data collected in the field is usually a richer source of information than laboratory data collected under controlled conditions. Field data takes into account the context in which the system is used.

We use "field test" to describe any version of software distributed to a group of people for use in their work. This definition includes the distribution of early subset versions as well as the later versions commonly referred to as field-test software. Early field testing often begins by giving a usable subset system to users who understand the status of the product and agree to use and evaluate it.

User visits, described previously, are a good way to collect field-test data. Another way to collect user feedback is by electronic communication. Digital's developers frequently use this effective method by making early field-test versions available on Digital's private world-wide DECnet network and by encouraging user feedback through electronic mail or a VAX NOTES conference.

Designers of the EVE text editor and VAXTPU software relied on user feedback by means of electronic communication throughout the development cycle. Preliminary versions of the EVE editor were available for daily work six months before external field test began. Overall, we received 362 suggestions from 75 different users. We implemented 212 (or 59 percent) of these suggestions for the version of EVE shipped with the VAX/VMS operating system version 4.2. We received 225 (or 62 percent) of the suggestions before field test began. More of these suggestions were implemented than suggestions received later: 65 percent of the suggestions received during internal field test were implemented compared to 48 percent of the suggestions received during external field test.

Although contextual interviews provide more information than users' reports of summary experience, the summary experience data is still valuable. The two methods complement each other. The on-site interviews provide details of users' ongoing experiences in the context of system use; on the other hand, electronic mail, conferencing, and problem reports provide summary experience data from a wider range of users than engineers could interview.

Early field testing is especially important for collecting data on experienced users. Experienced users, as well as new or infrequent users, must find systems easy to use. Early field testing is an excellent way to develop a test population of experienced users before a product is released. By the time later field test versions are available, these experienced users will be a valuable source of data on longer-term usability issues.

Instrumenting the System to Collect Usage Data

Knowing how frequently and in what order people use a system's functions helps engineers with low-level design decisions. For example, engineers can use usage data to order functions on menus, putting less frequently used commands on less accessible menus. Our group has collected and analyzed usage data for text editors and operating systems, and compared this with data collected by other groups.[13,14]

We collect usage data by asking people to use an instrumented version of a functioning system, either an existing product or a field-test version. We collect the most complete data by recording and time-stamping each individual user action. Keeping frequency counts of user actions also provides useful usage data, but does not include data on transitions between actions or time spent with different functions.

For the EVE editor, we used command frequency data from five different text editors to guide the initial design of the keypad interface and the command set. During internal field test, we collected command frequency data from a small set of EVE users to refine the command set. We also used command transition data as the basis for the arrangement of the arrow keys on the LK201 keyboard into an inverted-T shape. Usage data from an experimental text editor showed that the transition from the down-arrow key to the left-arrow key occurred more than twice as often as any other transition between arrow keys.[11,13] The inverted-T arrangement also allows three fingers of the user's hand to rest on the three most frequently used arrow keys, with an easy reach up to the up-arrow key.

Collectors of usage data must be concerned about user privacy and system performance. Users should know about the nature of the data collection and be informed when data is being collected. They should also have the option of using a system that has not been instrumented and does not collect usage data.

To inform users that data is being collected, designers can modify the instrumented version of the system so that a notification message is displayed each time this version is invoked. Users are thus reminded that all actions are being recorded. To minimize performance problems on instrumented versions, engineers can design the logging system so that any necessary delays occur at the start and finish of an application, not at random intervals while the application is being used.

Analyzing the Impact of Design Solutions

Designers make an impact analysis of user data collected during evolutionary delivery to estimate the effectiveness of design techniques in meeting product goals.[15] In usability engineering, design techniques are usually ideas developed after watching people use computer systems. Estimating the effectiveness of a set of design techniques for meeting a set of usability attributes helps to economically focus engineering effort on key issues.

Impact analysis tables contain percentage estimates of the contribution of each technique to the planned levels for each usability attribute. Impact analysis tables list product attributes and proposed design techniques in a matrix. Each entry in the table estimates the percentage that this technique will contribute toward meeting the planned level of this attribute.

Our software usability group creates impact analysis estimates in several ways, such as analyzing the videotapes made during user visits. With laboratory tests, we have derived estimates from the time actually spent as a result of interface problems encountered on a benchmark task.[16] Impact analysis data can also be presented graphically using Pareto charts.[17]

Conclusion

Our group applies usability engineering in the development of many new software products within Digital. Software usability engineering techniques can be used by any group of engineers that designs interactive software. No special equipment or prior experience is necessary to start applying these techniques, although equipment and experience can improve the results.

As we have gained experience with usability engineering, we have moved from laboratory tests to field visits as the main source of usability data. We find that field-test data provides a richer source of ideas for user interface design. Laboratory testing is still valuable, however, especially for testing early prototypes. We are now bringing some contextual interview techniques to our laboratory tests, interviewing users as they perform a task rather than observing them as they work on their own. For more advanced prototypes, we may ask users to use the system with their own work, which they bring with them to the laboratory. Controlled laboratory experimentation techniques are still useful for deciding some important design issues, such as choosing screen fonts for an application.

A user-oriented approach to software design requires a commitment to understanding and meeting users' needs through observation of people using systems. Software usability engineering techniques, applied in whole or in part, can produce computer systems that enrich human experience.

Acknowledgments

The techniques described here were developed in a group effort by present and past members of the Software Usability Engineering Group, including Mark Bramhall, Alana Brassard, Jim Burrows, Elisa del Galdo, Charles Frean, Kenneth Gaylin, Karen Holtzblatt, Sandy Jones, Thomas Spine, Eliot Tarlin, John Whiteside, Chauncey Wilson, Dennis Wixon, and Bill Zimmer. Dorey Olmer helped edit this paper.

References

1. T. Winograd and F. Flores, *Understanding Computers and Cognition: A New Foundation for Design* (Norwood: Ablex, 1986).

2. J. Whiteside, "Usability Engineering," *Unix Review*, vol. 4, no. 6 (June 1986): 22–37.

3. W. Deming, *Quality, Productivity, and Competitive Position* (Cambridge: MIT Center for Advanced Engineering Study, 1982).

4. T. Gilb, "Design By Objectives," Unpublished manuscript available from the author at Box 102, N-1411 Kolbotn, Norway (1981).

5. J. Bennett, "Managing to Meet Usability Requirements: Establishing and Meeting Software Development Goals," *Visual Display Terminals*, eds. J. Bennett, D. Case, J. Sandelin, and M. Smith (Englewood Cliffs: Prentice-Hall, 1984): 161–184.

6. P. Gilbert, "Development of the VAX NOTES System," *Digital Technical Journal* (February 1988, this issue): 117–124.

7. H. Dreyfus and S. Dreyfus, *Mind over Machine* (New York: The Free Press, 1986).

8. F. Brooks, Jr., "No Silver Bullet: Essence and Accidents of Software Engineering," *IEEE Computer*, 20, no. 4 (April 1987): 10–19.

9. M. Good, "The Iterative Design of a New Text Editor," *Proceedings of the Human Factors Society 29th Annual Meeting*, vol. 1 (1985): 571–574.

10. B. Curtis, et al., "On Building Software Process Models Under the Lamppost," *Proceedings of the IEEE 9th International Conference on Software Engineering* (1987): 96–103.

11. M. Good, "The Use of Logging Data in the Design of a New Text Editor," *Proceedings of the CHI '85 Human Factors in Computing Systems* (1985): 93–97.

12. M. Good, J. Whiteside, D. Wixon and S. Jones, "Building a User-Derived Interface," *Communications of the ACM*, 27 (October 1984): 1032–1043.

13. J. Whiteside, et al., "How Do People Really Use Text Editors?" *SIGOA Newsletter*, 3 (June 1982): 29–40.

14. D. Wixon and M. Bramhall, "How Operating Systems Are Used: A Comparison of VMS and UNIX," *Proceedings of the Human Factors Society 29th Annual Meeting*, vol. 1 (1985): 245–249.

15. T. Gilb, "The 'Impact Analysis Table' Applied to Human Factors Design," *Human-Computer Interaction—INTERACT '84*, ed. B. Shackel (Amsterdam: North-Holland, 1985): 655–659.

16. M. Good, et al., "User-Derived Impact Analysis as a Tool for Usability Engineering," *Proceedings of the CHI '86 Human Factors in Computing Systems* (1986): 241–246.

17. K. Ishikawa, *Guide to Quality Control*, second revised ed. (Tokyo: Asian Productivity Organization, 1982).

digital™

ISBN 1-55558-005-X