HEWLETT-PACKARD
JOURNAL

MARCH 1986

HEWLETT-PACKARD JOURNAL

March 1986 Volume 37 • Number 3

Articles

In this Issue

The software development process is being subjected to intense scrutiny and a lot of fine tuning these days. In HP's Corporate Engineering Department, the Software Engineering Laboratory leads companywide efforts in software metrics, tools, productivity, and environments, and has been holding an annual Software Productivity Conference. The papers presented at these conferences are full of creative engineering and good ideas. Taken together, they show that HP's software laboratories are investing an impressive portion of their resources in efforts to improve every aspect of the process of software development, an indication that the importance of software is being recognized as never before.

The original versions of the papers on pages 4 through 36 of this issue were presented at the 1985 HP Software Productivity Conference, and the subject of the article on page 37 was presented in a paper at the 1984 conference. Most of the papers deal with internal design, testing, and measurement tools and methods. On page 4, Marty Cagan of HP Laboratories presents a summary of HP's artificial intelligence workstation research efforts, which have so far produced one product, a Common Lisp development environment for the HP 9000 Series 300 workstation family. This product, along with some experimental software and the hardware to run it all, is being given, in a major HP grants program described on page 7, to several universities to aid their research in artificial intelligence and symbolic programming. The cover photo shows the HP Flight Planner/Flight Simulator, an application developed using HP AI workstation technology. The Flight Planner/Flight Simulator is described on page 13.

On page 43, you'll find a short article about the design of a new plug-in ROM package for the HP-71B Handheld Computer. The package makes it easier to program the HP-71B to control the HP 3421A Data Acquisition/Control Unit for low-cost, battery-powered, portable data acquisition and control applications.

-R.P. Dolan

What's Ahead

Scheduled for March are six articles on the design details of the HP 54100A/D and HP 54110D Digitizing Oscilloscopes. These general-purpose oscilloscopes are especially useful for digitial design and high-speed data communications applications. Also in the issue is an article on a software package designed to teach the fundamentals of digital microwave radio.

The HP Journal encourages technical discussion of the topics presented in recent articles and will publish letters expected to be of interest to our readers. Letters must be brief and are subject to editing. Letters should be addressed to: Editor, Hewlett-Packard Journal, 3000 Hanover Street, Palo Alto, CA 94304, U.S.A.

An Introduction to Hewlett-Packard's AI Workstation Technology

Here is an overview of HP artificial intelligence workstation research efforts and their relationship to HP's first AI product, a Common Lisp Development Environment.

by Martin R. Cagan

HEWLETT-PACKARD RECENTLY ENTERED the artificial intelligence (AI) arena with the announcement of its first symbolic programming product, the *Hewlett-Packard Development Environment for Common Lisp*. The technology underlying HP's initial product entry is the result of more than five years of research and development on what has evolved into the *Hewlett-Packard AI Workstation*. This article provides an overview of the AI Workstation technology.

The Hewlett-Packard AI Workstation represents the aggregate of the major symbolic programming software development efforts at Hewlett-Packard. (Previously, this research effort was internally referred to as the Prism program.) The term AI Workstation refers to the company-wide internal research and development program in AI, rather than to a particular product. In addition to the many HP divisions whose efforts have contributed key system components, many important concepts are based on research from the Massachusetts Institute of Technology (MIT), the University of California at Berkeley, and the Xerox Palo Alto Research Center (PARC). The University of Utah, in particular, has contributed significantly. Currently, HP's AI Workstation is actively used by well over two hundred people at various HP divisions, as well as by students and professors at major research universities across the United States. HP recently announced a $50 million grant of hardware and software which will provide Hewlett-Packard AI Workstations to selected major computer science universities (see box, page 7).

The AI Workstation technology is both portable and scalable, and can run on a variety of processors and operating systems, including the new HP 9000 Series 300 workstation family under the HP-UX operating system. The first and primary product that is an offspring of the AI Workstation technology is the Hewlett-Packard Development Environment for Common Lisp, announced at the 1985 International Joint Conference on Artificial Intelligence. Much of the technology described in this article is experimental and the reader should not assume the software discussed here can be purchased. Those components that are part of the Hewlett-Packard Development Environment for Common Lisp or other products will be noted.

There has been a great deal written in the press recently regarding symbolic programming technology and AI. The transition from numeric programming to symbolic programming is analogous to the "algebraization" of mathematics that occurred a century ago. The axiomatic, abstract algebraic viewpoint that was needed to simplify and clarify so many puzzles then is likened to the need for symbolic programming techniques to help solve today's difficult computational problems. AI applications such as natural language understanding, theorem proving, and artificial vision all rely on symbolic programming techniques for their flexibility and power in manipulating symbols, manipulating relationships between symbols, and representing large and complex data structures. The AI Workstation is a software system designed to solve problems using symbolic programming techniques. This article explores the AI Workstation by describing it from four perspectives: the market, the technology, the environment, and the applications.

The Market

There are many opinions concerning the future direction of the software market, but most agree that software is steadily becoming more complicated, powerful, and intelligent. Hewlett-Packard's AI Workstation provides the technology for developing and executing intelligent and sophisticated applications.

At Hewlett-Packard, AI techniques are viewed as an enabling technology. The AI Workstation provides tools and facilities that enable the programmer to create applications that were previously considered infeasible. These applications include expert systems, artificial vision, natural language interfaces, robotics, and voice recognition systems. Development and execution of these AI applications often require capabilities not available or feasible in conventional computer systems. For example, consider an expert tax advisor application. Such a system would need to embody the relevant knowledge and reasoning strategies of human tax advisors. AI-based techniques provide the necessary mechanisms for this knowledge representation and reasoning.

The AI Workstation's use need not be restricted to problems requiring the direct employment of AI technology, however. It has also been designed to foster improvements in the conventional software development market. For example, a typical tax accounting application may not need AI techniques for its implementation, yet can be implemented and maintained more productively by employing AI-based software development tools, such as tools that intelligently help locate and diagnose errors in the program code.

The AI Workstation is used by the software developer

to develop applications, and by end users to run AI-based applications. One of the AI Workstation's primary contributions to the AI market is that it provides both a development environment and an execution environment for AI applications, and it provides both on low-cost, conventional hardware such as the HP 9000 Series 300.

The software developer sees the AI Workstation as an environment tailored for the rapid development of software systems. The languages provided are geared for high productivity. The environment allows multiple programs, written in multiple languages, to be created, tested, and documented concurrently. Interpreters and compilers allow systems to be developed incrementally and interactively. The software developer can use the AI Workstation for the development of knowledge-based systems, or simply as a more productive means of generating conventional software written in conventional languages. In general, then, two reasons motivate the use of the AI Workstation as a software development machine. Either the AI Workstation technology is necessary to develop a particular AI application, or the user has a need to develop conventional applications in conventional languages more productively.

The end user of AI Workstation-based applications views the AI Workstation as an execution environment for applications that are highly interactive, intelligent, and customizable. The end user benefits from the total system, with high-resolution graphics, color displays, local area networks, multiple windows, and special-purpose input devices. The AI Workstation is modular and scalable so that a particular application can run with a minimum of resources and therefore keep the delivery vehicle's cost as low as possible. This is a major feature for many AI Workstation users who wish to both develop and distribute applications using the AI Workstation. To these users, providing a low-cost delivery vehicle is a major concern.

The AI Workstation also supports the notion of a server. A server is a system that is located on a network, dedicated to running a particular application. Other systems on the network, possibly even other servers, can send requests to the server to perform a function. The server performs the task and when appropriate, responds to the sender. AI Workstation-based servers and workstations make it easy for applications to create and send programs back and forth. A machine receiving a program is able to execute the program within the context of its local environment. Networks of servers running AI Workstation-based applications may prove to be a cost-effective solution for many users.

There has already been a great deal written about the large potential for productivity and quality improvements in software development, and given the rising cost of software development, there is a high demand for such improvements. Traditionally, AI researchers have demanded more productive and powerful software engineering tools from their environments. This was necessary to manage the scope and complexity of their software systems. Now that powerful, personal workstations are cost-effective, a wide range of software engineering tools, previously feasible only on expensive mainframes or special-purpose hardware, are now available for the design, development, testing, and maintenance of software systems.

The Technology

The evolution of the technology underlying the AI Workstation began with a joint development effort by HP Laboratories and the University of Utah.[1] The goal of this effort was to create a portable, high-performance implementation of a modern Lisp system so that programmers could enjoy efficiency and portability comparable to C and Fortran, along with the interactive and incremental program development and debugging environment of Lisp. Previously,

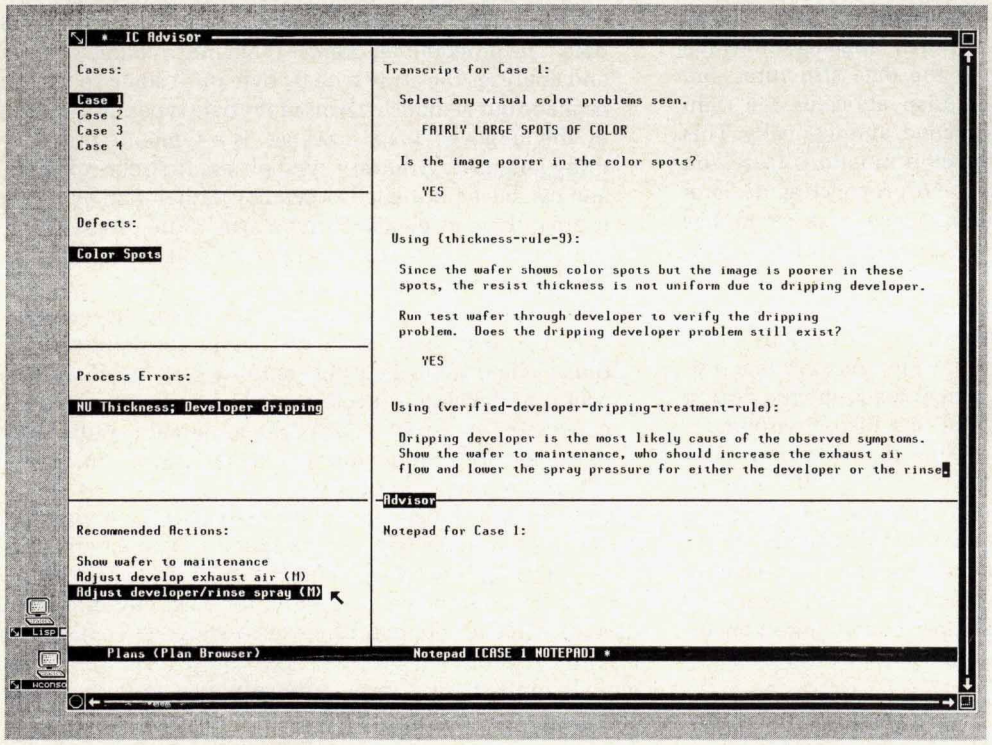


Fig. 1. *IC Photolithography Advisor screen showing symptoms, analysis, and recommended treatment. (See "Diagnostic Systems," page 12.)*

to enjoy high performance from Lisp, special-purpose, expensive hardware was required. A major contribution of the resulting underlying Lisp technology is that it is efficient even on conventional, low-cost hardware.

Lisp. Lisp is the dominant programming language for artificial intelligence research in the United States. But why Lisp? From a historical standpoint, Lisp is second in endurance and longevity only to Fortran. The modern Lisp systems, such as Hewlett-Packard's implementation of Common Lisp,[2] feature less cryptic alternatives to the basic Lisp commands, as well as many of the control structures and data types that have proven useful in conventional languages.* Although Lisp has evolved from its original form, it is for the most part as it was designed in 1958 by John McCarthy. Unlike Fortran, however, Lisp is attracting new converts daily, and is more popular today than it has ever been in its 28-year history. Unfortunately, many programmers in the industry today have not yet had the opportunity to work with Lisp as a production language, thus making it difficult to compare Lisp with C, Pascal, Fortran, or COBOL. A discussion of the primary features of Lisp follows, so that programmers of conventional languages can get an idea of what it is like to develop in a Lisp environment.

- Lisp supports incremental development. In conventional languages, when trying to build a program incrementally, the programmer must perform a number of time-consuming tasks, such as writing procedure stubs, including declarations, and constructing or simulating data. Each iteration requires an edit/compile/link/test cycle. In contrast, the Lisp programmer can simply write a function in terms of other functions that may or may not have been written yet and build either in a top-down fashion or in a bottom-up fashion, creating and testing continuously. The function can be executed as soon as it has been typed in.
- Lisp programs don't need declarations. Unlike C, Pascal, COBOL and most other conventional languages in which the programmer must specify the data structures and variables before using them, Lisp allocates the right amount of storage, when it is needed, automatically. This allows the programmer to develop functions truly "on the fly," without maintaining and propagating declarations throughout the program. Once a program has stabilized, the programmer can add declarations to improve the efficiency.
- Lisp provides excellent debugging. The Lisp environment supports an attitude towards error diagnosis that is quite different from that induced by conventional programming languages. When a bug is encountered during development of a Lisp program, the Lisp environment invites the programmer to explore the environment in which the exception was detected. The full power of Lisp itself is available to the programmer when debugging. Data structures can be analyzed and functions redefined. In fact, the programmer can even construct new Lisp functions on the fly to help diagnose the problem. In Lisp, a program error is less an error and more a breakpoint where the programmer can examine the system.
- Lisp manages memory automatically for the programmer. Memory management and reclamation are taken care of automatically in a Lisp environment. With conventional languages, memory management often accounts for a significant portion of the programmer's code. In Lisp systems, however, Lisp itself tracks memory use and reclaims unneeded storage automatically. This service allows the programmer to concentrate on the problem at hand, without having to manage the resources needed to implement the problem's solution.
- Lisp programs can easily create or manipulate other Lisp programs. Lisp is unique among major languages in that Lisp programs and data are represented with the same data structure. The benefits that result from this characteristic are many and have proven to be among the major contributions to the power of Lisp. This characteristic, for example, makes it is easy to write Lisp programs that create other Lisp programs, as well as to write Lisp programs that can understand other Lisp programs. Programs can be manipulated as data, and can be immediately executed or transferred to another Lisp machine for execution.*
- Lisp programs can run with a mix of compiled and interpreted code. The AI Workstation provides both a Lisp compiler and a Lisp interpreter. For development, the interpreter allows enhanced debugging and quick incremental design. Once a program is ready to be put into use, it can be compiled to increase its performance and reduce its code size. During development, however, the programmer often needs to run with a mix of compiled and interpreted code. The AI Workstation's Lisp has the feature of allowing an arbitrary combination of compiled and interpreted code. It is not unusual for a programmer to redefine compiled functions at run time to examine and explore the behavior of the application.
- Lisp is comfortable with symbols. In conventional languages, arbitrary symbols are treated as unstructured data. The programmer coerces them into a character array and analyzes the array byte by byte until some sense can be made out of them in terms of the data types understood by the language. Lisp, however, is a symbolic programming language. Arbitrary symbols are first-class objects, and can be manipulated as symbols rather than by trying to treat them as elements in an array. The programmer, in turn, can give symbols properties and manipulate relationships between symbols.
- Lisp is easy to extend. Functions defined by the programmer are treated in the same way as system-defined functions. When implementing complex systems, it is often useful to develop a specific vocabulary of functions for conversing in a particular problem domain. With Lisp, these specific, problem-oriented languages can be developed easily and quickly.

Because of its longevity and its many useful features, the reader may wonder why conventional programmers have not been using Lisp for years. There are three major reasons for this. First, until very recently, the Lisp environments described above were available only on large and expensive machines, and even on these machines, Lisp was using more than its share of resources. Only now, with the avail-

*Common Lisp, with the support of the U.S. Department of Defense, has emerged as the industry-standard Lisp dialect.

*"One person's data is another person's program."—Guy L. Steele, Jr.

HP's University AI Program

Lisp, like most of the AI technologies, was developed at a university. John McCarthy was at the Massachusetts Institute of Technology with Marvin Minsky when he developed the first Lisp version in 1958. Today, the universities of the world are still the primary source of basic research in AI.

To help universities in their AI research programs, Hewlett-Packard has implemented the largest single grants program in its history. In early 1985 we announced a program to grant up to $50 million in equipment to selected universities in the United States. In October, after receiving over 50 proposals, we selected 15 schools.

Each university will receive 20 to 60 engineering workstations, one year of support, and experimental AI software from HP Laboratories.

The schools we have selected are Brown University, the California Institute of Technology, Carnegie-Mellon University, Columbia University, Cornell University, the Massachusetts Institute of Technology, Stanford University, the University of California at Berkeley, the University of California at Los Angeles, the University of Colorado at Boulder, the University of Pennsylvania, the University of Southern California, the University of Texas at Austin, and the University of Utah.

The equipment will be used by faculty and graduate students to conduct research, and in classrooms for instruction.

By providing the latest equipment and software to these schools, we hope to see great advances in the state of the art in AI. In addition, we expect the universities to be able to train more AI practitioners, and train them better on the same caliber of equipment they will use on the job.

AI Research Configuration

The workstations granted to universities are fully configured to support advanced research. They include:

- HP 9000 Model 320 CPU using the Motorola 68020 processor with seven megabytes of RAM
- High-resolution color monitor
- Mouse
- Expander box
- 110 megabytes of hard disc storage (2 HP 7945 disc drives)
- Cartridge tape drive for system backup (HP 9144A)
- Local area network interface and software
- HP-UX operating system with windows and graphics utilities
- HP Laserjet Plus Printer (1 for each 10 workstations on the net)
- Plotter (HP 7550A) (1 for each 10 workstations on the net)
- Five additional disc drives (5 HP 7945s for 20 workstations).

Some researchers plan to use the AI Workstations for graphics research. They will receive the above configurations with the addition of the following:

- Graphics display station with eight planes of color and a second high-resolution color display
- Graphics accelerator.

AI Research at MIT

Three years ago Hewlett-Packard made a grant of five HP 9000 Model 236 Pascal workstations to Professor Gerald Sussman at the Massachusetts Institute of Technology. Using these systems, Prof. Sussman developed his own dialect of Lisp, called Scheme, which is oriented to the teaching of programming.

Today, students taking MIT's 6.001 class, the first course in the computer science curriculum, learn Scheme instead of Fortran or Pascal or C. With his colleague, Harold Abelson, Prof. Sussman has written a book, *The Structure and Interpretation of Computer Programs*. That book is now used to teach programming skills at Hewlett-Packard and at campuses across the U.S.A.

With his new grant, Prof. Sussman plans to take advantage of the skills the students learned in 6.001 to improve the curriculum for Signals and Systems, a required course for all electrical engineering majors. He proposes to put sophisticated simulations of circuits at the students' disposal. With their programming skills they can explore those simulations, develop their own experiments, and learn by doing. "For the first time," says Prof. Sussman, "we can incorporate into the curriculum explicit theories of how skilled engineers analyze and design technological artifacts. Our challenge now is to learn how to describe, explain, and teach the process of engineering."

Prof. Sussman's work in computer-aided-instruction is based on theories developed by Marvin Minsky and Seymour Papert. His research may lead us to new models of learning that will allow students to learn and retain more material more rapidly.

Seth G. Fearey
AI University Grants Program Manager
Hewlett-Packard Laboratories

ability of inexpensive, high-performance workstations and improved compiler technology, has Lisp become a cost-effective solution for conventional software development. Second, production languages were previously judged largely on the efficiency of compiled code. Now that the constrained resource is the software development cost rather than the delivery machine hardware, languages are being judged based on a different set of values.[3]

Third, while the features of Lisp described above are valuable, they do not come without a cost. Most Lisp systems remain ill-suited for such problems as real-time and security-sensitive applications. Reducing these costs is a major research topic at many university and industrial research laboratories. At HP, we have acknowledged the fact that different languages are optimized to solve different problems, and we have provided the ability for the Lisp environment to access arbitrary C, Pascal, and Fortran routines. This has important ramifications for HP and its customers. It is not necessary to discard existing code and data libraries to enjoy the benefits of Lisp. For example, an intelligent front end that accesses Fortran code libraries for instrument control can be written in Lisp. (The extensions to Common Lisp for foreign function calling are part of the Hewlett-Packard Development Environment for Common Lisp product.) AI Workstation-based applications are often blends of Lisp and conventional language components.

Object-Oriented Programming. The AI Workstation provides two higher-level languages, themselves implemented in Lisp, which support alternative paradigms for software development. The first of these language extensions supports object-oriented programming while the second supports rule-based programming.

HP provides a Lisp-based object-oriented programming language. (The extensions to Common Lisp for object-oriented programming are part of the Hewlett-Packard De-

velopment Environment for Common Lisp product.) Most of the AI Workstation's environment itself is written using this technology. Object-oriented programming is very much on the rise throughout the entire industry, and for good reason. Object-oriented programming brings to the programmer a productive and powerful paradigm for software development. It is a paradigm that addresses head-on the serious problems of code reusability and software maintainability by employing powerful techniques such as inheritance, data abstraction, encapsulation, and generic operations.[4]

Unlike most conventional languages, object-oriented Lisp is a language designed to support a particular programming methodology. The methodology, with support from the language, provides explicit facilities for code reusability, software maintainability, program extensibility, and rapid development.

The essential idea in object-oriented programming is to represent data by a collection of objects and to manipulate data by performing operations on those objects. Each object defines the operations that it can perform.[5]

The first facility I will describe is the notion of *data abstraction*. Using the object-oriented style of programming, each object is regarded as an abstract entity, or "black box," whose behavior is strictly determined by the operations that can be performed on it. In other words, the only way an object is accessed or modified is by performing the operations explicitly defined on that object. In particular, the internal data structure used to represent the object is private, and is directly accessed only by the operations defined on the object. Operations are invoked by sending *messages* to the object.

One advantage of the object-oriented style of programming is that it *encapsulates* in the implementation of an object the knowledge of how the object is represented. The behavior of an object is determined by its external interface, which is the set of operations defined on the object. If the designer changes the representation of an object, and the externally visible behavior of the operations is unchanged, then no source code that uses the object need be changed.

For example, suppose we wish to define a type dog. Using the object-oriented extensions to Common Lisp, our definition might be:

```
(define-type dog
  (:var name)
  (:var age)
  (:var owner)
  )
```

This says that we are defining a new type of object dog, with an internal representation consisting of a name, an age, and an owner. For example:

```
(setf fido (make-instance 'dog :name "Fido"))
```

This sets the variable fido to an instance of the type dog, with the name "Fido." If we wished, we could create one hundred instances of the type dog, each unique, whether or not they have the same name (just as there are many dogs, with more than one named "Fido"). Note that externally, nothing knows of our internal representation of the type dog. We could be implementing the dog's internal representation any number of ways.

We define operations on type dog by specifying the type and the operation, any parameters required by the operation, and the implementation of the operation. For example, to define an operation that will let us change the dog's owner:

```
(define-method (dog :give-new-owner) (new-owner)
  (setf owner new-owner)
  )
```

Note that the implementation of the operation is the only place where the internals of type dog are referenced. The value of this encapsulation is that if we decide to change the implementation of type dog, then it is only the type definition and the operations defined on that type that need to be modified.

We can access and manipulate the object by sending messages to it requesting it to perform specific operations. For example, to change Fido's owner to "Mandy":

```
(→ fido :give-new-owner "Mandy")
```

This statement reads, "Send the message to fido :give-new-owner 'Mandy'." Typically, we would define a number of operations for the type dog, such as sit, stay, come, and speak. These could then be invoked:

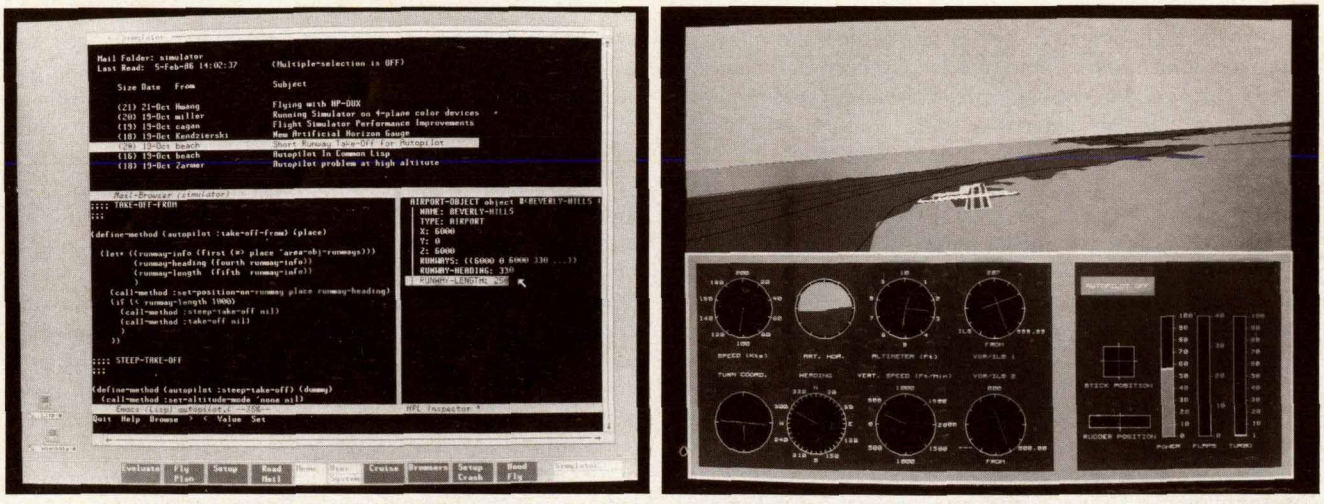


Fig. 2. *Flight simulator screens. Also see cover photo. See text on page 13 for details.*

```
(→ fido :sit)
(→ fido :stay)
(→ fido :come)
(→ fido :speak)
```

The second facility I will describe addresses the problem of *code reusability*. To a certain extent, the data abstraction facilities described above help ease the reuse of code modules in that the implementation is encapsulated, and the external interface is well-defined. More directly applicable to this problem is the concept of *inheritance*. A new type of object can be defined that inherits from other types. All of the operations that manipulate the types and the data maintained by the types are inherited. A new type definition can selectively override specific characteristics of the types that it inherits from. Thus, to define a new type that is only slightly different from some existing type, one might simply have the new type inherit from the existing type and override those aspects that differ in the new type. For example, to define a new type of dog, golden-retriever:

```
(define-type golden-retriever
  (:inherit-from dog)
  (:var number-of-tennis-balls-retrieved)
  )
```

This says that we want to define a new type golden-retriever, which inherits the data and operations from the type dog. In addition to the inherited attributes, we define golden-retriever's to maintain the attribute number-of-tennis-balls-retrieved. Note that when using an object, one cannot observe whether or not that object's type was defined using inheritance.

We create an instance of the new type golden-retriever:

```
(setf mac (make-instance 'golden-retriever :name "Mac"))
```

For this new type of dog, we would have our own implementation of the :speak operation, one that produces a deeper bark than the inherited version. We would also have some additional operations defined which are appropriate only to objects of the type golden-retriever. For example, we have the additional operation fetch, which of course is an attribute of all retrievers, but not all dogs, as well as the new operation :make-coffee (Mac is a very smart dog). These could then be invoked:

```
(→ mac :speak)
(→ mac :fetch)
(→ mac :make-coffee :time 0700)
```

Note that we could have made further use of inheritance by first defining a type retriever that inherited from type dog, and then defining the new types golden-retriever and labrador-retriever which inherit from the type retriever.

Another facility provided by object-oriented Lisp is the support of a powerful form of *generic operations* known as *polymorphism*. When one performs an operation on an object, one is not concerned with what kind of object it is, but rather that an operation is defined on the object with the specified name and the intended behavior. This ability is lacking in languages like Pascal, where each procedure can accept only arguments of the exact types that are declared in the procedure header. As an example of the value of generic operations, suppose one day we attempt to replace Man's Best Friend with a robot, presumably one domesticated to the same extent as a dog is. We could implement the new type robot as follows:

```
(define-type robot
  (:var name)
  (:var model)
  (:var owner)
  )
```

To create an instance of type robot:

```
(setf roby (make-instance, 'robot :name "Roby"))
```

Suppose that we have an existing library of applications that currently direct objects of the type golden-retriever in various tasks. If we were to implement the same functional operations performed by objects of type golden-retriever for the type robot, then all of our application code would work unchanged:

```
(→ roby :sit)
(→ roby :stay)
(→ roby :come)
(→ roby :fetch)
(→ roby :speak)
(→ roby :make-coffee :time 0700)
```

Note that while the implementations of these operations differ, the functional specification and the external *protocol* for dealing with objects of type golden-retriever and robot are defined to be the same, so our applications work unchanged, and we save on dog food, too.

The facilities of object-oriented programming described here can go a long way towards improving program maintainability, program extensibility, and code reusability. Object-oriented programming has been used to implement operating systems, window managers, market simulations, word processors, program editors, instrument controllers, and games, to name just a few of its applications. Its paradigm has proven productive, powerful, and easy to learn and use.

Rule-Based Programming. The second of the alternative paradigms provided in the AI Workstation is the Hewlett-Packard Representation Language (HP-RL), HP's experimental rule-based programming language.[6] HP-RL is intended to support the development of knowledge-based software systems. Knowledge-based software systems, which include expert systems, are systems that search a *knowledge base* of information and attempt to make deductions and draw conclusions using the rules of logical inference. A knowledge base is a data base that embodies the knowledge and problem-solving strategies of human experts. In an expert system, there is rarely a procedural description defined in advance for solving a problem. The system must search the knowledge base and make inferences by using the rules and strategies defined by the developer. Current knowledge-based software systems include applications such as medical consultation systems, integrated circuit diagnostic systems, tax advisors, and natural language understanding systems.

The key to knowledge-based systems lies in representing the vast amounts of knowledge in an organized and manageable structure. Without such organization, problems quickly become intractable. An intractable problem is one that cannot be solved in a reasonable amount of computation time. HP-RL provides data structures and control structures specifically for knowledge representation, knowledge organization, and reasoning about that knowledge.

HP-RL allows knowledge to be represented as *frames*. A

frame is a data structure that groups together arbitrary amounts of information that are related semantically.[7] Typically, a frame is used to store information specific to, or about, a particular entity. HP-RL allows knowledge to be organized into frames of related information. Like object-oriented programming, HP-RL provides the ability for frames to inherit information from other frames. For example, a frame that describes a specific entity such as a person, Jane, might inherit characteristics from related entities such as scientist and female. Therefore, the entity Jane automatically inherits all of the attributes of females and scientists. Attributes specific to Jane can then be specified to differentiate Jane from other female scientists.

Frames can be grouped into domains of knowledge. This sort of partitioning reduces problem complexity, and can also improve the efficiency of searches through the knowledge base by helping the program avoid searching through irrelevant knowledge. Searching through the knowledge base is a sophisticated process performed by the HP-RL *inference engine*. The inference engine is the facility that scans the knowledge base trying to satisfy rules. Rules in HP-RL are frames composed of a set of premises and conclusions, similar to an if-then construct in conventional languages. HP-RL provides both forward chaining and backward chaining rules. The inference engine applies forward chaining, or data driven, rules to infer conclusions given verified premises. The inference engine applies backward chaining, or goal driven, rules to find verifiable premises, given a desired conclusion.

As an example, consider a rule that says: If a dog is a golden retriever, then the dog likes tennis balls. If we define the rule to be a forward chaining rule, then when the inference engine is searching the knowledge base, if the current data supports the assertion that the dog is a golden retriever, then we can infer that the dog likes tennis balls. If we define the rule to be a backward chaining rule, then when the inference engine is searching the knowledge base, if the desired goal is to find a dog that likes tennis balls, then the inference engine will check to see if the current data supports the assertion that the dog is a golden retriever.

One of the primary differences between rule-based approaches and conventional programming is that in rule-based programs, the program's flow of control is not explicit in the program. The process of deciding what to do next is consciously separated from data organization and management. The programmer can help direct searches by using *heuristics*. A heuristic is a rule that guides us in our navigation and search through a knowledge base. Managing searches through the knowledge base is a major research topic, since an intelligent and selective search of a knowledge base can make the difference between a usable system and an unusable system. Searching the knowledge base is where most of the computing resources are spent when using a knowledge-based system. To help with this problem, HP-RL provides for the incorporation of heuristics about dealing with other heuristics, which can be used to govern the strategy of the program and therefore conduct searches more intelligently.

HP-RL currently contains a number of experimental facilities which are being studied and tested to discover more effective mechanisms for performing the difficult task of capturing and using knowledge.

The Environment

One of the primary differences between programming with Lisp and programming with other languages is the environment provided for the programmer. The AI Workstation provides access to all data and execution via an integrated environment. The user environment is unusually flexible and powerful. It contains a large and powerful

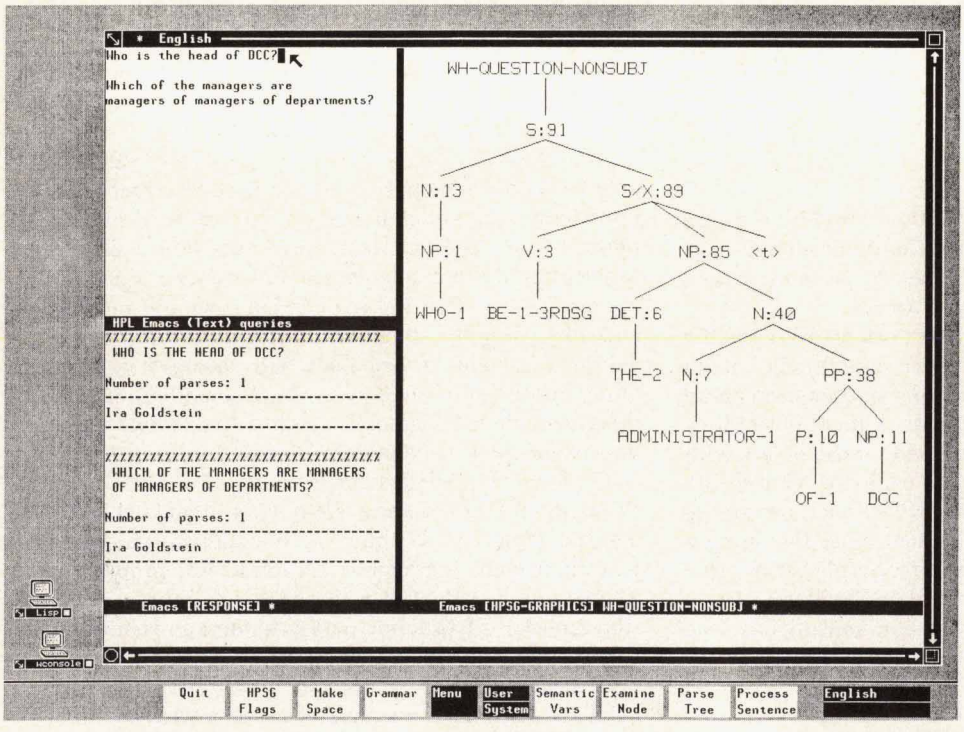


Fig. 3. *Natural language understanding system screen, showing an English query, a parse tree showing how the system interpreted the sentence, and the results of the query.*

collection of text-manipulation functions and data structures useful in constructing user interfaces, text and graphics editors, and browsers. The following sections explore these various components of the AI Workstation user environment.

Editing. The AI Workstation environment contains a version of EMACS, an editor originally developed by Richard Stallman at MIT. Hewlett-Packard's object-oriented Lisp implementation of EMACS, like the original MIT EMACS, is a customizable, extensible, self-documenting, screen-oriented, display editor.[8]

Customizable means that users can mold the AI Workstation EMACS in subtle ways to fit personal style or editing tasks. Many user-level commands exist to allow the user to change the environment's behavior dynamically. Customization is not limited to programmers; anyone can easily customize the environment.

Extensible means that the user can make major modifications and extensions to the AI Workstation EMACS. New editing commands can be added or old ones changed to fit particular editing needs, while the user is editing. The user has a full library of text-manipulation functions at hand for the creation of new editing functions. This type of extensibility makes EMACS editors more flexible than most other editors. Users are not constrained to living with the decisions made by the implementors of the AI Workstation EMACS. If the user has a need for a new function, or a reason to modify the behavior of an existing function, then the user is able to make the modification quickly and easily.

Self-documenting means that the AI Workstation EMACS provides powerful interactive self-documentation facilities so that the user can make effective and efficient use of the copious supply of features.

Screen-oriented means that the user edits in two dimensions, so the page on the screen is like a page in a book, and the user has the ability to scroll forward or backward at will through the book. As the user edits the page, the screen is updated immediately to reflect the changes made. Just as many books on a desk can be open and in use at once, with the AI Workstation EMACS, many screens can be visible and active simultaneously. In fact, one of HP's extensions to MIT's EMACS is the ability not only to have multiple screens active on a single physical display, but also to have multiple screens on multiple physical displays. (The EMACS-based editing environment described here is part of the Hewlett-Packard Development Environment for Common Lisp product.)

Browsing. Another feature of the AI Workstation user environment is a large library of tools known as browsers. Browsers are more than an integral component of the user environment; they are a metaphor for using the environment. A browser is a simple tool for the convenient perusal and manipulation of a particular set of items.[9] Experimental browsers in the AI Workstation environment include documentation browsers, file browsers, mail browsers, source code browsers, and application browsers. These browsers range from simple to very complex. Users can list all the mail messages sent by a particular person regarding a particular subject, or can instantly retrieve the definition of a particular Lisp function. The user can conduct automated searches of the documentation, or can browse and manipulate the contents of a complex data structure.

Browsers provide a simple, intuitive, integrated interface that is useful for handling a wide range of problems. The environment provides a library of browser construction tools and functions to allow users to create their own browsers for their particular applications and needs.

Programming. On the AI development machine, a large portion of the user environment is tuned to support the programming task, which includes activities such as program editing, debugging, testing, version and configuration management, and documentation. The AI Workstation supports development in Lisp, C, Pascal, and Fortran. In addition, a toolkit is provided to let users customize the environment for other languages. The AI Workstation provides an integrated and uniform model of multilingual software development.

One of the major features of the AI Workstation user environment is the interface to the underlying Lisp system. Lisp programmers enjoy direct access to the Lisp compiler and interpreter without having to leave the environment. This means that a program can be edited, tested, debugged, and documented incrementally and interactively as the program is developed. The editing is assisted by an editor that understands the syntax of Lisp. Testing is assisted by Lisp interface commands, which pass the text from the program editor to the underlying Lisp system and return the results back to the environment. Debugging is assisted by an interactive debugger, function stepper, and data inspector available directly from the environment. Program documentation is assisted by documentation tools designed for the programmer which generate much of the formatting details automatically.

Using the foreign function calling facilities of the AI Workstation described earlier, non-Lisp programmers can also enjoy many of the benefits of interactive, incremental development. For example, the AI Workstation contains full two- and three-dimensional vector and raster graphics operations. (The graphics facilities referred to are provided on the HP 9000 Series 300 running under the HP-UX operating system.) While these operations are C routines, all are directly accessible from the Lisp environment. Typically, C programmers must iterate through the edit/compile/link/test cycle as they develop a graphics application. In contrast, using the AI Workstation, C programmers can step through the development of their graphics applications statement by statement, and enjoy immediate feedback simply by observing the results on the screen. Once the program is functionally correct, the programmer can convert the statements into a formal C program, and compile it with the standard C compiler.

Managing. The AI Workstation user environment contains a variety of optional service applications to support the programmer in dealing with office and management functions. Experimental applications developed with this technology include electronic mail, project management, documentation preparation, slide editing, calendar, spreadsheet, information management, and telephone services. Each of these applications, once the user chooses to include it, becomes an integral part of the environment. Because all of these applications are written using the the AI Workstation environment facilities, they are customizable, extensible,

and accessible from anywhere in the environment. For example, the user can move from creating a slide to reading a mail message to testing Lisp code and back to creating the slide.

Interfacing. The AI Workstation's user environment contains tools that greatly simplify the incorporation of new input and output devices such as tablets, touchscreens, or voice synthesizers. In addition to supporting standard keyboard and mouse input, experimental versions of the AI Workstation environment also support joystick, tablet, touchscreen, videodisc, voice input and output, and touchtone telephone input. The user environment also supports many user interface models, and provides a library of environment functions to help users define their own user interface model. Existing user interface models include pop-up menus, softkeys, English commands, and **CONTROL-META** key sequences.

The AI Workstation does not impose a particular interface model on the user. Default interfaces exist, but the user is free to modify or add any user interface desired. Delivery applications written to run under the AI Workstation environment can choose to use one or more of the supplied user interfaces, or the designer can define a new interface.

The Applications

This section examines some of the primary types of applications the AI Workstation technology was designed to develop and run. Note that unless specified otherwise, these applications are experimental and not available for purchase.

Diagnostic Systems. Diagnostic systems are good examples of expert system applications. Diagnostic systems retrieve as much data as possible from instruments and/or users, and attempt to determine the cause of the problem and/or the remedy. Diagnostic system applications include medical diagnostic systems, instrument diagnostic systems, and intelligent computer-assisted instruction applications.

At HP Laboratories, we are experimenting with an IC photolithography diagnosis system (see Fig. 1, page 5). This system, called the Photolithography Advisor, is an expert system used to diagnose failures in the negative photolithography resist stage of IC fabrication.[10]

Within Hewlett-Packard's computer support organization, a number of diagnostic expert systems are employed. The Schooner expert system diagnoses and corrects data communication problems between a terminal and an HP 3000. The AIDA expert system provides an efficient tool for analyzing HP 3000 core dump files. The Interactive Peripheral Troubleshooter system diagnoses disc drive failures.

Instrument Control. A growing class of expert systems deals with the intelligent control, monitoring, and testing of instruments, as well as the interpretation of the data gathered by these instruments. Instrument control and interpretation applications include network analysis, factory floor monitoring, process control, and many robotics applications.

At Hewlett-Packard, one experimental application helps with the interpretation and classification of data collected by a mass spectrometer. Another application analyzes data from a patient monitoring system. Within the AI industry, a number of intelligent instrument and process control applications are being developed, such as a system that monitors the operations of an oil refinery.[11]

Simulations. Many complex software systems fall into the category of simulations and modeling. Simulations play major roles in nearly every aspect of a business. The object-oriented programming facilities discussed earlier enable engineers to program simulations rapidly. Simulation applications include econometric modeling, flight simulation, chemical interaction modeling, and circuit simula-

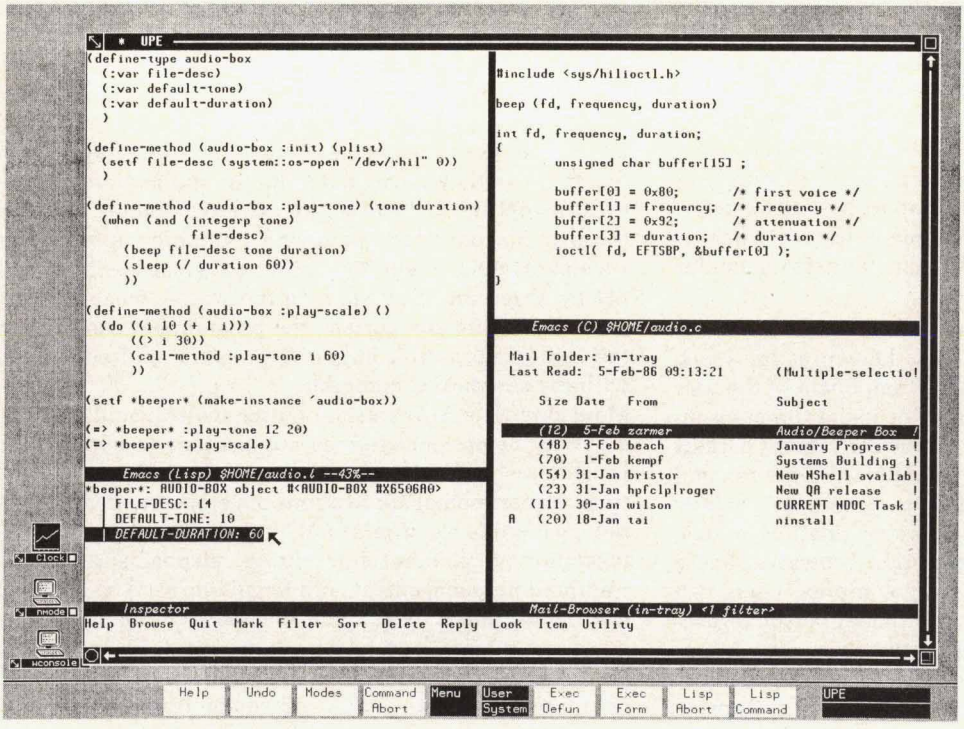


Fig. 4. *Unified programming environment screen shows multilingual support with simultaneous development in C and Lisp, integrated mail, and dynamic data inspection.*

tions.

At HP Laboratories, for example, we have implemented VLSI logic simulators, which enable an engineer to design, debug, test, and evaluate circuit designs before incurring any actual manufacturing expense.[3]

The HP Flight Planner/Flight Simulator (see Fig. 2 and cover) is an application designed by HP Laboratories to illustrate a number of important features of the AI Workstation technology: namely, that multilingual applications are desirable and simple to develop, that complex applications can be developed rapidly, that Lisp applications can be designed to run without the interruption of garbage collections, and that Lisp applications can run on conventional hardware and operating systems at very high performance.

The Flight Planner module is a constraint-driven expert system for planning a flight. The system presents a detailed map of California stretching from San Francisco to Los Angeles. The pilot is asked for an originating airport, a final destination, and any intermediate stops desired. The pilot then is allowed to specify specific constraints, such as "Avoid oceans and mountain ranges," "Ensure no longer than 3 hours between stops," or "Plan a lunch stop in Santa Barbara." The system's knowledge base includes data on the airports, the terrain, and the specifications and capabilities of a Cessna 172 airplane. With the constraints specified, the Flight Planner attempts to find a viable flight plan that satisfies the constraints specified by the pilot, as well as the constraints implied by the limitations of the terrain and aircraft.

Once a flight plan has been generated, the Flight Planner passes the flight plan off to the Flight Simulator module, which then flies the plan as specified. The flight plan specifies the destination, route, and cruise altitude for each leg of the flight. The flight simulator's autopilot module, using these directions as well as the specific airport and airplane data from the knowledge base, performs the takeoff, flies the plane using ground-based navigational aids, and executes an instrument landing. In addition to flying predetermined flight plans via the autopilot, the Flight Simulator can be flown manually. The pilot uses an HP-HIL joystick, a 9-knob box, and a 32-button box as the controls.

The Flight Planner is implemented using HP-RL. The Flight Simulator is implemented in Common Lisp and the object-oriented extensions to Common Lisp. The graphical transformations are performed by C routines accessed from Lisp, using the 3D graphics facilities of the HP-UX operating system. The model of flight, the autopilot component, and the scene management are all written using the object-oriented extensions to Common Lisp.

The Flight Simulator required two months for two people to develop, while the Flight Planner required a month for three people.

Natural Language. With the computational and reasoning capabilities of systems such as the AI Workstation, computational linguists are making headway into the difficult field of natural language understanding. At HP Laboratories, computational linguists have been using the AI Workstation to develop an experimental, domain independent, natural language understanding system (see Fig. 3). HP's natural language system employs a hierarchically structured lexicon, a set of lexical rules to create derived lexical items, and a small set of context-free phrase structure rules as the data structures used in parsing English sentences and questions. Interpretations of these sentences are the result of the meanings of the individual words together with the semantic rules that are associated with each of the dozen or so phrase structure rules. What the natural language system produces is a set of unambiguous application independent expressions in first-order logic, each expression corresponding to one possible interpretation of the original

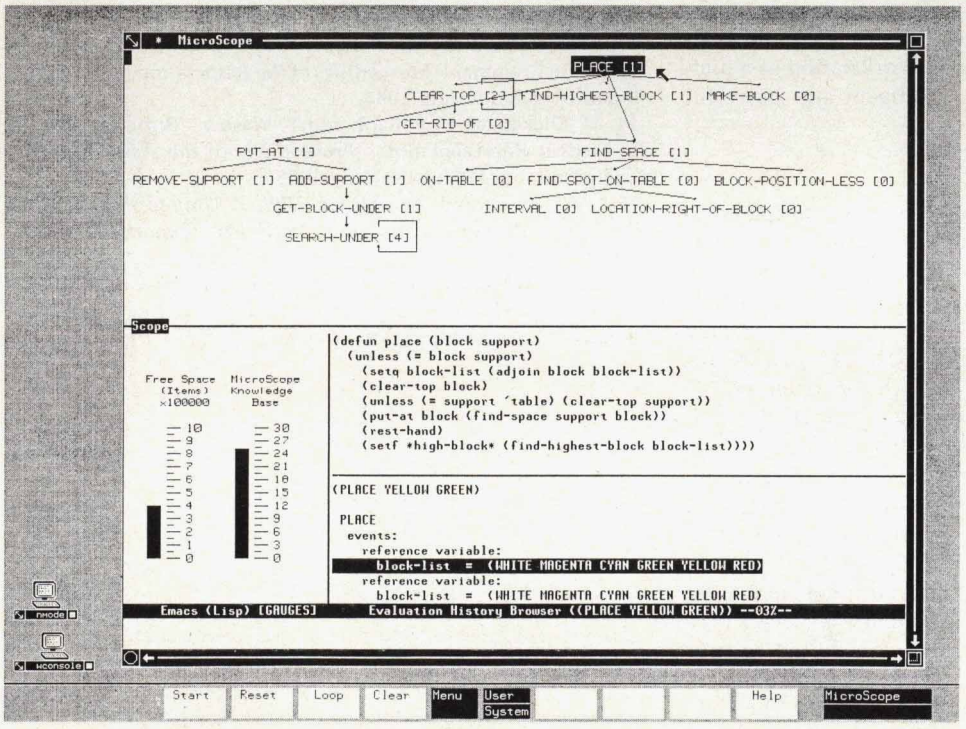


Fig. 5. *A screen from MicroScope, a knowledge-based program analysis tool, showing a call graph monitoring program execution, the source code of the highlighted module, evaluation history browser, and system resource monitors.*

sentence. In test applications, these expressions are transduced into either data base queries or messages to objects, making use of the domain-specific knowledge in each application to make precise those relations or pronoun bindings that were underspecified in the sentence itself.[12,13,14]

Software Engineering. While environments such as the AI Workstation can significantly improve software productivity, we are just beginning to reap the benefits of applying AI to the software development process itself. There are a number of projects throughout the industry working in this area.

At HP Laboratories, we are working on intelligent programming environments that help the user assess the impact of potential modifications, determine which scenarios could have caused a particular bug, systematically test an application, coordinate development among teams of programmers, and support multilingual development in a uniform manner (see Fig. 4).[15] Other significant software engineering applications include automatic programming, syntax-directed editors, automatic program testing, and intelligent language-based help facilities.

Applying AI to the software development process is a major research topic.[16] There is tremendous potential for improving the productivity of the programmer, the quality of the resulting code, and the ability to maintain and enhance applications. One of HP's first projects of this type is MicroScope, a tool to help software engineers understand the structure and behavior of complex software systems (see Fig. 5).

Conclusion

We have discussed the AI Workstation from the point of view of the software market, the underlying technology, the user environment, and the AI-based applications. Having studied the AI Workstation from each of these perspectives, we hope that the reader will assimilate this into a coherent and accurate view of the HP AI Workstation technology. Over the coming years, HP engineers and our partner universities will be using the AI Workstation as a platform for exploring increasingly intelligent and powerful applications and technologies.

Acknowledgments

Since the AI Workstation is defined to be the aggregate of HP's research in the AI area, the efforts of well over 100 people at Hewlett-Packard divisions and universities around the United States are represented. Major contributions came from Martin Griss and his Software Technology Laboratory, the Knowledge Technology Laboratory, the Interface Technology Laboratory, and the director of HP Laboratories' Distributed Computing Center, Ira Goldstein. The Fort Collins Systems Division, with teams led by Roger Ison and John Nairn, provided an existence proof to the computer industry of a high-performance, quality implementation of Common Lisp on conventional hardware. The Computer Languages Laboratory developed the extensions to the AI Workstation for conventional languages. The faculty and students of the University of Utah, Professor Robert Kessler in particular, contributed greatly to the fundamental capabilities of the AI Workstation. A number of consultants from Stanford University, the University of Utah, the Rand Corporation, and the University of California at Santa Cruz continue to help us improve our technology. This article has benefited from the insights of Ralph Hyver, Seth Fearey, Martin Griss, and Alan Snyder of HP Laboratories, and Mike Bacco and Bill Follis of Fort Collins. The author also thanks Cynthia Miller for her long hours of editing.

References

1. M.L. Griss, E. Benson, and G.Q. Maquire, "PSL: A Portable LISP System," *1982 ACM Symposium on LISP and Functional Programming*, August 1982.

2. G.L. Steele, *Common Lisp: The Language*, Digital Press, 1984.

3. J.S. Birnbaum, "Toward the Domestication of Microelectronics," *Communications of the ACM*, November 1985.

4. M. Stefik and D.G. Bobrow, "Object-Oriented Programming: Themes and Variations," *The AI Magazine*, January 1986.

5. A. Snyder, *Object-Oriented Programming for Common Lisp, HP Laboratories Technical Report ATC-85-1*, February 1985.

6. S. Rosenberg, "HP-RL: A Language for Building Expert Systems," *Proceedings of the Eighth International Joint Conference on Artificial Intelligence*, August 1983.

7. R. Fikes and T. Kehler, "The Role of Frame-Based Representation in Reasoning," *Communications of the ACM*, September 1985.

8. R.M. Stallman, "EMACS: The Extensible, Customizable, Self-Documenting Display Editor," in Barstow, Shrobe, and Sandewall, *Interactive Programming Environments*, McGraw-Hill,1984.

9. A. Kay, "Computer Software," *Scientific American*, Vol. 251, no. 3, September 1984.

10. T. Cline, W. Fong, and S. Rosenberg, "An Expert Advisor for Photolithography," *Proceedings of the Ninth International Joint Conference on Artificial Intelligence*, August 1985.

11. R.L. Moore, L.B. Hawkinson, C.G. Knickerbocker, and L.M. Churchman, "A Real-Time Expert System for Process Control," *Proceedings of the 1984 Conference on Artificial Intelligence Applications*, December 1984.

12. C.J. Pollard and L.G. Creary, "A Computational Semantics for Natural Language," *Proceedings of the Association for Computational Linguistics*, July 1985.

13. D. Proudian, and C. Pollard, "Parsing Head-Driven Phrase Structure Grammar," *Proceedings of the Association for Computational Linguistics*, July 1985.

14. D. Flickenger, C. Pollard, and T. Wasow, "Structure-Sharing in Lexical Representation," *Proceedings of the Association for Computational Linguistics*, July 1985.

15. M.L. Griss and T.C. Miller, *UPE: A Unified Programming Environment, HP Laboratories Technical Report STL-85-07*, December 1985.

16. D.R. Barstow and H.E. Shrobe, "From Interactive to Intelligent Programming Environments," in Barstow, Shrobe, and Sandewall, *Interactive Programming Environments*, McGraw-Hill, 1984.

A Defect Tracking System for the UNIX Environment

Created in response to a lack of effective defect tracking and analysis tools for software development, DTS is now used by 24 HP Divisions.

by Steven R. Blair

DTS IS A DISTRIBUTED defect tracking system that simplifies the process of reporting, collecting, and summarizing software defect data. It provides utilities for submitting, receiving, resolving, and archiving defect reports, and for generating detailed and summary report listings.

DTS was designed to serve the defect tracking and metrics collection needs of prerelease software development. It is available only in HP software development laboratories. It runs under versions of AT&T Bell Laboratories' UNIX™ operating system on networks of HP 9000 and Digital Equipment Corporation VAX™ computers.

DTS was created in response to a lack of defect tracking and analysis tools in software development environments. For example, when a defect report was submitted, it often didn't describe the problem adequately or contain the information necessary to reproduce the problem. There was also no easy way to tell if the defect report got to the person who had to research and fix the problem, and it was a difficult and tedious task for a manager to get an accurate count of defects for a lab, a project, or a particular engineer. Finally, data that managers needed for development process metrics wasn't always collected.

DTS solves these problems in a way that is easy to learn and use. This paper presents the DTS solution first from the project management perspective and then in terms of user interaction. It describes the system's operational environment and shows the current status of DTS use at HP.

Managing Software Development with DTS

DTS supports software development management by collecting and organizing defect data, and by automating the tracking of submitted and resolved defects. DTS collects data that both lab and project managers can use to answer questions about their software development processes. Examples of this data are the submitter's name, the name of the responsible project, the module that is defective, the date the defect was found, the symptoms caused by the defect, and the severity of the defect. See the Appendix for a complete list of the data items kept for each defect. This information can be grouped, summarized, and used for evaluating a product's development.

Several graphical examples of summarized DTS data are given in Figs. 1 through 4. Although DTS does not currently generate graphs, it does provide the defect data that other tools can use to generate graphs.

DTS provides defect information at the lab and program level. Data from DTS general summary reports can be used to generate a mean time between failures (MTBF) graph (Fig. 1) and a defect arrival rate graph (Fig. 2) for either a single project or a group of projects.

For project managers, DTS provides summaries and synopses of project-level defect information. This data can be used to generate a defect resolution rate graph (Fig. 3) and a defect backlog graph (Fig. 4).

DTS also provides project and engineer summary defect

Fig. 1. *Mean time between failures graph.*

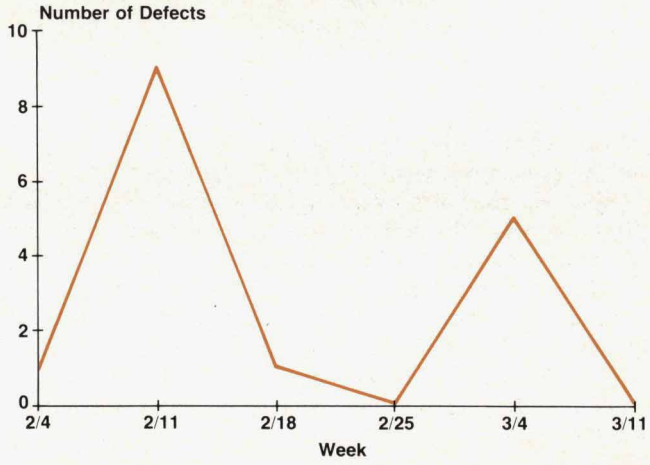


Fig. 2. *Defect arrival rate graph, showing total number of defects reported each week.*

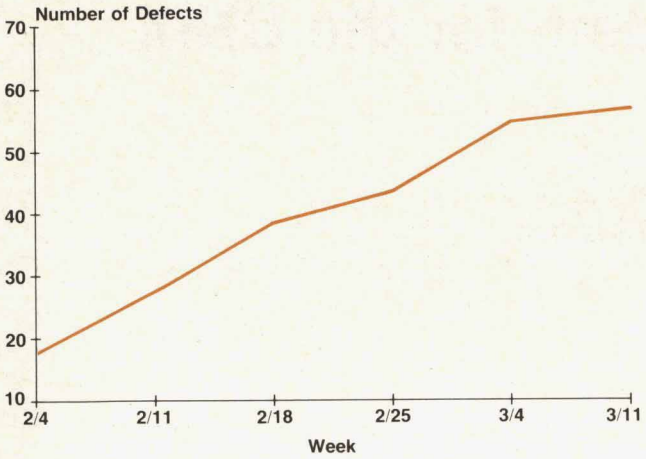


Fig. 3. *Defect resolution rate graph, showing defects resolved as a function of time.*

lists. These lists give a two-line synopsis for each resolved, open, or unreceived (submitted to, but not yet accepted by a project) defect. These synopses can be organized either by project or by engineer assigned to fix the defect. Fig. 5 is an example of a project summary report for project dts.

Managers will probably want to view other defect data relationships in addition to the examples shown above. The DTS report generator, DTS administrator utilities, and UNIX tools can be used to extract, organize, and summarize defect data from the ASCII-text defect reports (see Appendix). This summarized data can then be formatted as desired for textual or graphical display.

Submitting Defects

Four DTS utilities allow a user to submit, receive, resolve, and summarize defect reports. A common user interface for all utilities makes DTS easy to learn and operate.

When a defect is found in a piece of software, the user submits a defect report by using the dtssub command at the user's workstation. DTS will then prompt the user for the data necessary to create a defect report. This includes:

- The submitter's name, phone number, electronic mail address, and project name
- The date the defect was found
- The name and version of the defective software
- The severity and urgency of the defect
- A one-line description of the problem
- The activity used in finding the defect
- The project responsible for fixing the defect.

The user may also attach up to twenty-six related text files to describe the problem further, present a workaround, document a fix, or provide other information.

After a defect report has been submitted, DTS moves it through the network to the machine that the responsible project has chosen to have its DTS defects sent to. When the incoming defect report arrives, DTS sends an electronic mail notification to the manager and engineers responsible for the project.

Receiving Defects

New (unreceived) defects that have arrived for a project are examined with the dtsrec command. The data for each unreceived defect is displayed, and the user is able to either (1) receive the defect report and accept responsibility for fixing it, (2) forward the defect report to another responsible project, or (3) push it back into the unreceived queue for later viewing. If the defect report is received, DTS prompts the user for:

- The priority assigned to the defect report
- The name of the engineer responsible for fixing it
- An estimated fix date.

After the defect is received, an acknowledgment is sent back to the submitter, and the defect report is incorporated into the DTS defect data base on the receiver's machine.

Updating and Marking Defects as Resolved

After a defect has been researched and resolved, dtsupd is used to update and mark the defect report as resolved. At that time the user needs to provide:

- The type of resolution (e.g., code change, design change)
- The name of the module that was fixed (when applicable)
- The amount of engineering time it took to fix the defect
- The development cycle phase in which the error was introduced, found, and fixed.

DTS notifies the submitter by electronic mail when the defect is resolved, and sends a copy of the defect report to the machine designated as the DTS archive for the site.

Summarizing via DTS Reports

The dtsrep command generates any of five different reports, each giving different levels of detail, and each aimed at different types of users. They are:

- General summary. Shows global totals for unreceived, open, resolved, and urgent defects for selected groups of projects. One likely grouping of projects might be a lab summary, listing totals for all the projects that make up a particular lab.
- Manager summary. Shows a two-line description for each of the defects belonging to a particular project,

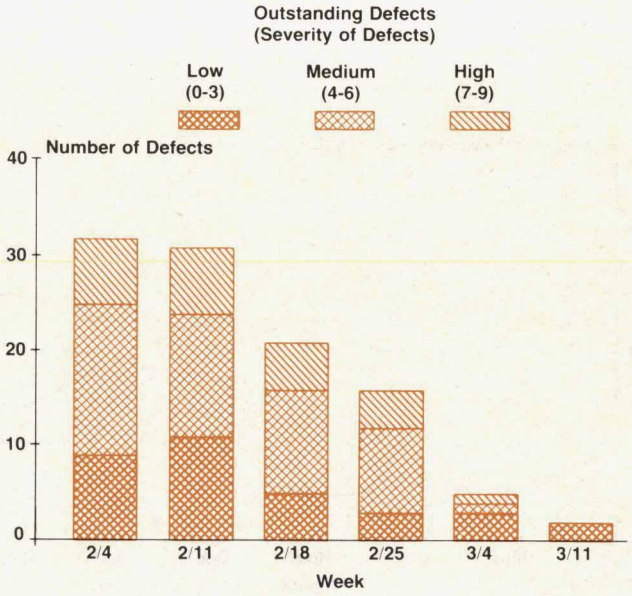


Fig. 4. *Defect backlog plot.*

```
Project Summary of Defect Information for: dts

Total Showstopper Defects:    0
Total Unreceived Defects:     0
Total Open Defects:         102
Total Resolved Defects:     312
Total Defects:              414

Engineer Defect Information:

Steven Blair

CLLab00160  Submit Num:  00034DSDsa  OPEN  S:3 P:6    Recvd: 841008
Dsc:  I type control-D at "finding activity" and dtssub quits on me

CLLab00165  Submit Num:  00031DSDsa  OPEN  S:2 P:6    Recvd: 841008
Dsc: Hitting ^D as response to resp_proj prompt in forwarding unchanges ui

DSDsa00032  Submit Num:  00008SELbb  OPEN  S:2 P:4    Recvd: 841029
Dsc: "dtssub -q another" does only one defect and quits without asking
```

Fig. 5. *An example of a project manager summary report.*

sorted by responsible engineer.

- Engineer summary. Shows a two-line description for each of the defects assigned to a particular engineer to fix.
- Unreceived summary. Shows a two-line description for each of the unreceived defects.
- Engineer details. Shows all of the available information about a single defect report.

Each of these reports can be viewed interactively, or sent to a file or printer to capture data for later analysis.

User Interface

Each of the four DTS utilities uses a common user interface that provides on-line help, three different modes for novice, intermediate, and expert users, and easy-to-define user default values.

The user interface makes DTS easy to learn and use. For example, when a DTS utility is being run, two forms of help are always available. One is a description of the field the user is prompted for, and the other is what data is appropriate for that field. While responding to a prompt, the user can hit the **ESC** key to obtain the field description. To determine what data is appropriate for that field (e.g., whether it is any string of characters or a specific selection from a list of items), the user types **CTRL G**. These help descriptions can be tailored for any specific site.

Since users vary in their familiarity with DTS from first-time to accomplished users, and therefore vary in how much detail they want in DTS prompts, there are three levels of prompting detail that can be selected: verbose (maximum information at each prompt), terse (minimum information at each prompt), and medium verbosity (which lies somewhere in between). All prompts in DTS are site configurable.

Finally, any data field DTS prompts the user for can be preset to a default value. For example, the user's name, phone number, and electronic mail address can be defaulted for the defect submission (dtssub) utility. This relieves the user of having to retype these items each time a defect is submitted. More subtly, default values allow DTS commands such as dtssub to be integrated into automated test suites. This can provide reporting of defects discovered during testing, without direct human intervention.

DTS Operational Environment

DTS is currently supported on HP 9000 Series 500 and 200 Computers running the HP-UX operating system, and on DEC VAX Computers running the 4.2 BSD operating system.

The DTS network is divided into sites, which are clusters of machines running DTS software. Each site has a site hub machine that acts as a gateway for information between other site hubs and machines on the site (Fig. 6). This network topology fits in with the network that has evolved at HP consisting of machines running versions of the UNIX operating system. A DTS site typically corresponds to a divisional site in this network.

There are three kinds of information that flow between DTS machines. The first kind is the defect report that travels through the network as it moves from the submitter's machine to the receiving project's machine. Next, project summary data is collected from each machine's local defect data base and is distributed to all other DTS machines. Finally, transaction acknowledgments are sent via electronic mail when the state of a defect report changes (e.g., when a defect report arrives, is received, or is resolved).

Occasionally there are data transmission problems and a defect report isn't able to get through, or there are network configuration problems and DTS doesn't know where to

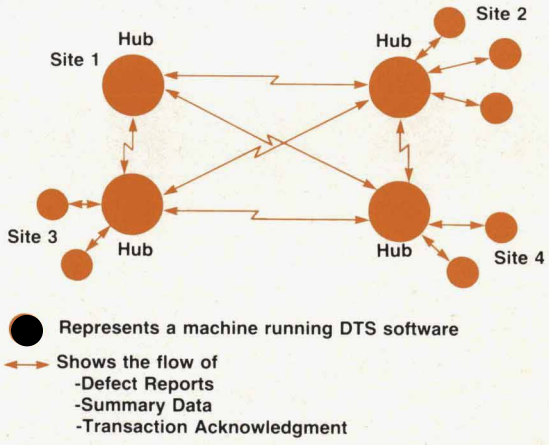


Fig. 6. *DTS network layout.*

send a defect report. In these cases DTS moves the data into a holding area and sends an electronic mail message describing the problem to the DTS site administrator.

The DTS site administrator is a person who has been identified as the DTS support person for a site. This person defines the DTS intrasite network topology, handles DTS installation, keeps DTS configuration files current, and coordinates local site customizations to DTS. After the initial setup time, the administrator can expect to spend 2 to 10 hours per month maintaining DTS.

The initial disc space required for DTS is three megabytes. This number will grow during use as defects are submitted and received. For example, people submitting and receiving five defect reports per day for two projects on a machine can expect DTS disc space requirements to grow an additional megabyte per month (assuming an average defect report length of 4000 bytes). Off-line data archiving can be used to cut this number in half.

DTS Today

DTS was released internally November 1, 1984 to software developers in three HP Divisions. Since then DTS has spread to other HP Divisions that develop software in the UNIX environment. Its user community has grown from 25 to over 200 users located at more than 24 HP Divisions. Because of this growing level of acceptance, Corporate Engineering's Software Engineering Laboratory has identified DTS as an important software development and management tool, and has committed resources to its support, promotion, and evolution.

Acknowledgments

Seven people from three different divisions made up the initial DTS product team. Dave Decot did a tremendous job developing an extensible user interface for DTS and writing the defect packet manipulation library. Steve Banks wrote the dtssub, dtsupd, and dtsrep user utilities and served as technical project lead. The author developed the packet transport software and the dtsrec user utility. Julie Banks worked very closely with the DTS development engineers and was key in keeping the fast-track DTS project on track. Debra Martin, Bob Grady, and William Woo also contributed management support. Thanks to Barbara Scott, Debbie Caswell, and Jack Cooley for their contributions to the DTS product design. Thanks also to Craig Fuget for his diligence in product testing.

Appendix
Contents of a Defect Report

This table contains the length in bytes, name, and description of each data field in a DTS defect report.

Length	Name	Description
10	Defect Number	Assigned when a defect is received
1	Defect Status	New, open, or resolved
10	Submitter Number	Assigned when the defect is submitted
20	Submitter Name	Name of the person who submitted the defect
20	Submitter Phone	The submitter's phone number
20	Submitter Address	The submitter's electronic mail address
20	Test System	The machine the defect was found on
20	Submitter Project	Name of the submitter's project
6	Date Found	Date the defect was found
20	Software	The name of the software that is suspected of being defective
10	Version	Version of the suspect software
1	Severity	The submitter's estimation of the defect severity
1	Showstopper	"y" if this defect is keeping a project from meeting a critical checkpoint
72	Description	Seventy-two-character defect description
2	Activity Used	How the submitter found the defect
4	Fixing Time	How long it took to isolate, fix, unit test, and document the defect fix
20	Responsible Project	The project that is responsible for fixing the defect
20	Responsible Engineer	The engineer assigned to fix the defect
8	Resolution	How the defect report was resolved
1	Priority	This defect's fix priority
6	Resolve Date	Estimated date of resolution, or date the defect was resolved
40	Fixed Module	The module or documentation changed
10	Related Defect	The number of a duplicate or related defect
6	Date Received	The date the defect report was received
1	Phase Introduced	The software development lifecycle phase in which the defect was introduced
1	Phase Found	The software development lifecycle phase in which the defect was found
1	Phase Fixed	The software development lifecycle phase in which the defect was fixed
2	Times Reported	The number of duplicates of this defect
21	Symptoms	The symptoms of the problem
1	Workaround	"y" if a workaround to the defect exists
2	File Count	The number of related files attached to this defect report
100	Reserved	Data space reserved for DTS
100	Unused	Additional data space available for sites to use
???	Related Files	Up to 26 related files may be attached to any defect report

A Toolset for Object-Oriented Programming in C

Object-oriented programming seeks to encapsulate entities in a program into objects, methods, and messages. It is useful for writing highly dynamic software that is well-structured and easily maintainable. This paper presents a set of tools that support object-with-methods data structuring.

by Gregory D. Burroughs

THE DATA STRUCTURES of a program are often composed of groupings of objects. For example, in describing a map, one might have city as one class of objects. Related to each city are groups of objects: residents, natives, streets, parks, significant buildings, and so on. In the design of algorithms that operate on the map, it would be useful to be able to group natives of a city into a structure without worrying about the details of the structure's implementation. One notation might be:

```
A city is {
        Name
        A Group "street" of streets
        A Group "residents" of persons
        A Group "natives" of persons
        ...
}

A person is {
        Social Security Number
        Grouped by city of residence
        Grouped by city of birth
}
```

As the design of algorithms progresses, appropriate data structures for implementing the groupings can be determined. If reports that list a city's residents and natives sorted by Social Security Number are required, some variety of tree might be used to implement the resident and native relationships. A convenient notation might be:

```
A city is {
        Name
        Tree "residents" of persons
        Tree "natives" of persons
        ...
}

A person is {
        Social Security Number
        Node "residents" in a tree of persons
        Node "natives" in a tree of persons
}
```

with the concepts of *grouping* and *grouped in* implicit in the notations Tree and Node. By this point in the design, the algorithms for manipulating persons relative to cities would be well-defined and, given the proper set of software tools, much of the program could be generated automatically. This paper describes a set of tools for generating objects and methods from class relationships as described above, and presents results of its use.

Objects

In an object-oriented program, data is organized into *classes*. A class contains its members, called *objects*, and operations for manipulating its members, called *methods*. Objects can be thought of as the data record for the class and methods as the functions that act on the data records. Cities and persons are among the classes known to the above example. Oakland is likely to be an object in the city class, and city-add-new-resident is likely to be a method for objects in the city class.

One advantage of object-oriented programming is the ability of each class to hide its data representation from other classes. Such separation is supported through the concept of *messaging*. In traditional programming languages, objects communicate by passing data structures to functions. This allows and even encourages methods of one object to use the structure of objects they reference. In a message-based system, objects communicate by passing messages to other objects, with the recipient object determining which method is appropriate for performing the requested action. One envisions the dynamics of a well-coordinated, properly staffed and trained team; each member performs assigned tasks, requesting assistance from other team members when necessary, but not meddling in the tasks of others.

Another advantage of object-oriented programming comes from the independence of representation and action. An object can ask for an action without worrying about the particular type of the object that will do the action. Since it is the recipient that determines how to respond to a message, the requester can use the same message and the same transmission mechanism to request a particular action (for example, print yourself) from objects of different classes.

Finally, objects can be *trained*. Actions appropriate at one stage in a program's execution might not be the same actions that were appropriate at an earlier stage. An object can change the method it will use for a given message to reflect its current state. The object-method-message paradigm has been found to be effective not only within an individual program, but also between concurrent, cooperating programs.[1,2]

In particular, the method paradigm is appropriate for abstract data structures. Often, application programs find that their jobs consist of creating and traversing data structures. Most standard texts present data structures and their methods (e.g., additions, deletions, merges, traversals) independently of any particular source of data.[3,4,5] That is, most operations on a data structure can be viewed as a method that manipulates just the structure and sends messages to the objects grouped in the structure when application specific information or actions are required. For example, the report that lists all residents of Oakland could be generated by the tree-traverse method walking the persons tree for the Oakland object and sending print messages to each person object it encounters.

When data groupings are viewed in this object-methods-message framework, program generation can proceed automatically from data structure design. In the example, once a tree has been selected to implement a grouping, the actions appropriate to creating and traversing the tree are also known and, in an ideal environment, should require little or no time or programming effort to implement.

Implementation

The following sections describe a toolset for automatically generating data records and methods from a description at the grouping level as described above. An implementation of the above example is used to explain the use of the package. The resulting C program source appears in the Appendix. Fig. 1 diagrams the use of the toolset.

The toolset is built on top of the C programming language using C-preprocessor macros and UNIX™ utilities. There

UNIX is a trademark of AT&T Bell Laboratories.

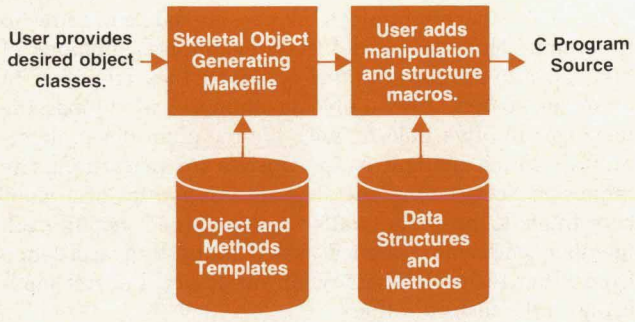


Fig. 1. *This figure diagrams the use of the object-oriented programming toolset. First, the user interactively defines the object classes desired. The skeletal object generator generates skeletal objects and methods. These consist of class initialization, object allocation, skeletal manipulation methods, and skeletal structure definitions. The user then adds appropriate manipulation and structure macros to the manipulation methods and structure macros. The resulting source is presented to the C compiler.*

are two types of macros: macros that generate data structure entries in a data record and macros that generate functions to manipulate the data structures. The UNIX utility make(1) is used to generate a template and rudimentary methods for each data class. In the examples that follow, C program syntax is loosely followed. The reader is directed to the bibliography for references on C and the UNIX operating system.[6,7,8]

A *dispatch table* is the mechanism that allows an object to map a message to the appropriate method. When a message is received, the object receiving the message extracts a method selector from the message, then uses the selector as a key into the dispatch table to determine which method to invoke. A dispatch table can support dynamism in three places. First, the method returned for a given selector can vary over the life of the object. Second, the number of selectors and methods in the table can vary. Third, the table can support various levels of indirection in the lookup mechanism. The dispatch table for the toolset of this paper supports the first level of dynamism by providing a static number of run-time-modifiable selector-to-method mappings for all classes of objects. Methods provided for each object include self-identification, self-printing, and comparison with another object in the same class. The object class has two methods, object-new and object-free, for allocating and deallocating objects in the class. The structure used to implement an object's dispatch table is a record:

```
struct methods {
        int         *(*new)();
        int         (*free)();
        int         (*key)();
        int         (*print)();
        int         (*compare)();
};
#define METHODS struct methods *methods
```

Each object class is represented by a structure. The macro METHODS provides each object in the class with a direct link to its associated methods:

```
struct <object> {
        METHODS;
        /* object specific slots */
};
```

The macros MESSAGE and MESSAGE2 are used to send a message to an object:

```
MESSAGE (sally, print)
```

calls the print method associated with sally and provides sally as the argument to print, while

```
MESSAGE2 (sally, compare, joe)
```

calls sally's compare method with sally and joe as arguments. Each object class is represented by a pair of files:

<object>.h <object>.c

The .h file contains the structure specification for the

object, while the .c file contains its methods.

Pseudoautomatic class generation is possible from this well-defined specification. A script make-object creates a .h file containing a skeletal structure description and a .c file containing rudimentary methods. For each class, make-object generates a function that initializes the method table and a function that calls the initializers for all of the classes.

Data Structure Macros

Data structures consist of two parts. The first part describes *features* required in the records making up the structure. The second part consists of *actions* appropriate to the data structure (insert, delete, traverse, union, ...). This toolset addresses both aspects of data structure implementation.

The part of a data structure linking one data record to another forms a *skeleton*. Typically, the skeleton consists of two types of pointers: the *head* of the structure and *members* of the structure. Skeletons are of two forms, *endogenous* and *exogenous*.[4] Endogenous skeletons link objects through fields inside the object's data record, while exogenous skeletons have a separate structure which points to objects from the outside. Endogenous skeletons tend to be used in groups that are all the same data type, since the data structure pointers are part of the data record definition. Methods for endogenous skeletons can be associated with the data structure itself, rather than with each object. Exoskeletal data structures are more suited to groupings spanning several data types, since the data type need not know the types with which it is grouped. In this case, methods must be associated with the objects. The greater flexibility of exoskeletal structures comes at a price of more declarations and somewhat larger data storage requirements. In the example, while methods remain associated with the objects, endoskeletal structures are used to reduce the number of declarations.

Macros used to create data structures reside in files named:

```
<data structure>.s.h
```

where the suffix .s.h stands for structural header. As an example, the macros for creating an AVL tree[9] are:

```
/* macro to generate an AVL tree root */
#define AVL_ROOT(root_type, tree_head_slot)
        struct {
                struct root_type        *root;
        }       tree_head_slot

/* macro to generate an AVL tree node */
#define AVL_NODE(tree_type, tree_thread_slot)
        struct {
                struct tree_type *lchild; *rchild;
                int             balance;
        }       tree_thread_slot

#define BALANCED                0
#define BALANCED_LEFT           -1
#define BALANCED_RIGHT          1
```

In the example, these macros could be used to create either an exoskeletal or an endoskeletal data structure. The declarations:

```
struct  city_type       {
        METHODS;
        string name;
        AVL_ROOT(person_type, natives_head);
        AVL_ROOT(person_type, residents_head);
                ...
}

struct  person_type     {
        METHODS;
        string SSN;
        AVL_NODE(person_type, natives_thread);
        AVL_NODE(person_type, residents_thread);
}
```

would generate an endoskeletal AVL tree structure. Note that the class declarations are not cluttered with the details of the record structure needed to implement the tree.

Macros used to create methods reside in files named:

```
<data structure>.m.h
```

where the suffix .m.h stands for manipulation header. Some of these macros manipulate the record structure required by the data structure—for example, the macro to initialize a node before insertion in an AVL tree:

```
#define AVL_NODE_INIT(node_obj, tree_thread)
        node_obj→tree_thread.lchild=NULL;
        node_obj→tree_thread.rchild=NULL;
        node_obj→tree_thread.balance=BALANCED
```

When information specific to the objects in the data structure is needed, the macros use the methods associated with the objects. For example, the following macro checks whether a node is already present in an endoskeletal AVL tree:

```
#define AVL_CHECK
        (tree_root_obj, tree_root,
        found_obj, found_obj_type, tree_thread,
        found_flag)
{
        struct found_obj_type   *marker;
        /*
         * here should be checks to insure that the tree
         * root exists and the like
         */
        found_flag = false;
        marker = tree_root_obj→tree_root.root;
        while (marker EXISTS)  {
            switch (MESSAGE2 (found_obj,
                compare, marker))  {
            case  EQUAL:
                found_flag = true;
                marker = NULL; /* please exit */
                break;
            case  LEFT:
```

```
                    marker = marker →
                        tree_thread.lchild;
                    break;
                case RIGHT
                    marker = marker →
                        tree_thread.rchild;
                    break;
                default:
                    FAULT;
                }
            }
}
```

In the example, the function city-has-resident? could be generated as follows:

```
int city-has-resident (city, person)
    struct    city_type     *city;
    struct    person_type   *person;
{
    int          found;
    AVL_CHECK (city, residents_head,
        person, person_type, residents_thread,
        found);
    return (found);
}
```

Consistent programming style relieves some of the clutter resulting from inclusion of type names in the parameter list, as does use of meaningful type and slot names.

Conclusion

The toolset provides a variety of data structures, most having their origin in a particular application where the toolset was used. Among the structures are basic data structures (singly and doubly linked lists with methods for stacks, queues, dequeues, and insertion sorted lists, binary trees with traversals, iterators), height-balanced trees (AVL trees),[9] self-adjusting trees (splay trees),[10] and k-d trees.[11] New structures can often be created by modifying existing ones.

Three programs represent the types of software developed using the toolset. TEST-BED is a test bed for the investigation of digital network traversal algorithms. VFORMAT is a discrete-event simulation time queue. EXPANDER is a hierarchical electrical circuit expansion program. Experience suggests that the tools are most useful in programs where the software author can naturally cast the application into the object paradigm and where data organization and traversal account for most of the programming task.

TEST-BED was written as an environment for exploring structural test generation algorithms for combinational circuits.[12] Initial object and data structure design required two engineering days. Input data formats were borrowed from the interactive logic simulator, ILS,[13] and a naive input reader. Rank ordering and signal propagation algorithms were quickly implemented. When it was realized that the initial data structures would not support all the traversals desired by some of the test generation algorithms, new structures were easy to add to the skeleton. Use of this toolset allowed implementation to proceed quickly and directly from data organization with the benefit that more investigation time was available to concentrate on algorithm and data structure design and subsequent algorithm performance analysis.

VFORMAT was written as a specific niche design tool for a specific Hewlett-Packard logic design team. With such a narrow scope, the design goals of VFORMAT emphasize functionality and speed of implementation rather than performance and enhancability. However, use of the toolset allowed data organization design to proceed assuming that efficient data structures could be developed as readily as inefficient structures that merely worked. The result was a quickly developed tool that performed considerably better than the minimum expectations.

EXPANDER manages an electrical circuit description used to communicate information between several programs. Most of EXPANDER's actions involve a traversal of a hierarchy of *blocks* and *instances* that describe circuit connectivity and parameters. The object-methods-message implementation allows traversal control to be implemented and debugged entirely independently of traversal actions. Each traversal of the hierarchy is regarded as an instance in the class of traversals whose actions are guided by a control object. Before a traversal starts, a traversal object is instantiated. This object is provided with a dispatch table of methods it will use to perform its specific algorithm. Then, at each potentially interesting location in the traversal, the control object sends a message to the traversal object to perform the action appropriate for this location in the hierarchy (e.g., process-a-child, return-from-processing-subtree) or to obtain the next location for processing (e.g., select-a-child, get-next-sibling).

EXPANDER demonstrates the toolset's utility on large, complicated programs. There are twenty object classes known to EXPANDER, some of these group or are grouped in as many as ten other object classes. Structure-generating macros make the organization and implementation more readable. An objects-methods-messages implementation allowed data, action, and control to be considered, implemented, and debugged separately. This separation eases the tasks of adding and modifying functionality and increases confidence in the program.

Directions for Further Study

The toolset presented in this paper was started before the general availability of commercial preprocessors and languages that directly support object-method-messages programming. Its primary goal was the support of quick implementation of data-structure-intensive programs. While it achieved its goals, initial writing and debugging of long macros requires a bit more effort than general C programming. Special makefile entries and the like make the job a bit less demanding, but the existence of commercially available languages whose definitions support objects-methods-messages (such as Common Lisp) suggest that program development should proceed in those languages.

This package has proved its usefulness in the rapid prototyping of data-structure-intensive programs. However, the program designer still needs to determine which structures are appropriate for the application at hand from either training or experience. An expert system could be de-

veloped to assist selection of appropriate groupings given the desired operations between object classes. When a grouping is described, structure declarations for the objects and method functions for the data structures could be generated automatically. This could then be coupled with a graphical interface for describing and documenting the object classes, their groupings and the operations.

Acknowledgments

I would like to acknowledge the team that worked on the Interactive Logic Simulator project where this work began—Ravi Apte, Antony Fan, G.A. Gordon, Jim Hsu, Kathy Hsu, Greg Jordan, Mark Millard, Viggy Mokkarala, and especially Bob Floyd—for their contributions.

References

1. T. Baker, "Tutorial in Objects and Modules in HP Pascal," *Proceedings of the HP Software Productivity Conference*, 1984.
2. G. Burroughs, "A Message-Based Methodology for Integrating CAE Tools," *Proceedings of the HP Software Productivity Conference*, 1984.
3. A. Aho, et al, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, 1974.
4. R. Tarjan, *Data Structures and Network Algorithms*, SIAM Publications, 1983.
5. N. Wirth, *Algorithms + Data Structures = Programs*, Prentice-Hall, 1976.
6. B. Kernighan and D. Ritchie, *The C Programming Language*, Prentice-Hall, 1978.
7. *UNIX Programmer's Manual, Vol. 2ab*, Bell Laboratories, 1979.
8. *UNIX Programmer's Manual, Vol. 2c*, University of California at Berkeley, 1979.
9. G. Adelson-Velskii and Y. Landis, "An Algorithm for the Organization of Information," *Doklady Akademiya Nauk SSSR*, Vol. 146, pp. 263-266 (Russian).
10. R. Tarjan, "Amortized Computational Complexity," *SIAM Journal of Algebraic & Discrete Methods*, Vol. 6, no. 2, 1985, pp. 306-318.
11. J. Bentley, "Multidimensional Binary Search Trees Used for Associative Searching," *Communications of the ACM*, Vol. 18, no. 9, 1975, pp. 509-517.
12. M. Breuer and A. Friedman, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, 1976.
13. G. Jordan, et al, "ILS—Interactive Logic Simulator," *Proceedings of the Design Automation Conference*, 1983.

Appendix

The following program fragment is the example from the accompanying paper as it would appear to the program designer.

```
#include        "constants.h"
#include        "methods.h"
#include        "AVL.m.h"

struct    city_type          {
          METHODS;
          string    name;
          AVL_ROOT(person_type, natives_head);
          AVL_ROOT(person_type, residents_head);
};

struct    person_type        {
          METHODS;
          string    SSN;
          AVL_NODE(person_type, natives_thread);
          AVL_NODE(person_type, residents_thread);
};

int city-has-resident (city, person)
          struct    city_type          *city;
          struct    person_type        *person;
{
          int            found;
          AVL_CHECK (city, residents_head, person, person_type,
                    residents_thread, found);

          return (found);
}
```

The following is the program fragment after processing by the preprocessor. Typically, the program designer would never look at this fragment.

```
struct    city_type          {
          method    *methods;
          string    name;
          struct {
               struct    person_type        *root;
          }
          natives_head;
          struct {
               struct    person_type        *root;
          }
          residents_head;
};

struct    person_type        {
          method    *methods;
          string    SSN;
          struct {
               struct    person_type        *lchild,
               *rchild;
               int            balance;
          }
          natives_thread;
          struct {
               struct    person_type        *lchild,
               *rchild;
               int            balance;
          }
          residents_thread;
};

int city_has_resident (city, person)
struct    city_type                    *city;
struct    person_type                  *person;
{
          int       found;
          {
               struct    person_type    *marker;
               found = false;
               marker = city → residents_head.root;
               while (marker EXISTS) {
                    switch ((*person → methods).
                         compare(person, marker)) {
                    case    EQUAL:
                         found = true;
                         marker = NULL;
                         break;
                    case    LEFT:
                         marker = marker →
                              residents_thread.lchild;
                         break;
                    case    RIGHT:
                         marker = marker →
                              residents_thread.rchild;
                         break;
                    default:
                         FAULT;
                    }
               }
          };
          return (found);
}
```

Tools For Automating Software Test Package Execution

Developed by one HP Division and now used by others, these two tools reduce the time it takes to develop test packages and make it easy to reuse test packages in regression testing.

by Craig D. Fuget and Barbara J. Scott

TWO SOFTWARE TESTING TOOLS in use at HP's Data Systems Division are the Virtual Terminal and the Scaffold Test Package Automation Tool and Test Package Standard.

The Virtual Terminal tool runs on the HP 125 Computer and is used to automate interactive testing of HP 1000 Computers. It simulates keyboard input to the host system and saves the input and output for comparison with master result files. It is also useful for testing non-forms-mode programs on HP 3000 Computers (the HP 125 hardware doesn't support block mode).

The Scaffold Test Package Automation Tool and Test Package Standard are used to create and run test packages. The Scaffold provides tools for test package creation and setup, and for running the tests and verifying the results. The Test Package Standard consists of documentation standards for test plans, test packages, and individual tests. The Scaffold was originally developed using the HP-UX operating system and has been ported to the HP 1000.

Virtual Terminal

The Virtual Terminal tool (VT) was created to allow automated regression testing of the interactive features of HP 1000 Computers. It has been used in cases where normal test automation is impossible (e.g., screen-oriented interactive programs). We have found that this improves regression test accuracy because each replay of the test causes exactly the same data to be entered. We have also found this to be a great productivity aid because the test engineer only needs to type the test data once.

Since all of the system's output is saved each time the program is run, it is simple to track changes in the system by using file comparison tools, thus automating result verification. This improves test accuracy.

Test automation is broken into two parts: the initial run where the engineer instructs the tool on how to test the system, and succeeding runs where the tool repeats what it was told. The two tasks are handled by the programs XMS and XVT, respectively.

Before the initial run, the interactive tests are carefully planned to ensure that all of the functions of the system under test (SUT) are covered. Once this is done, the SUT is prepared with the latest revision of the software to be tested.

To set up for the run, the engineer attaches the HP 125 to the test system, and then runs XMS. Because XMS is transparent to both the user and the SUT, the tests can be

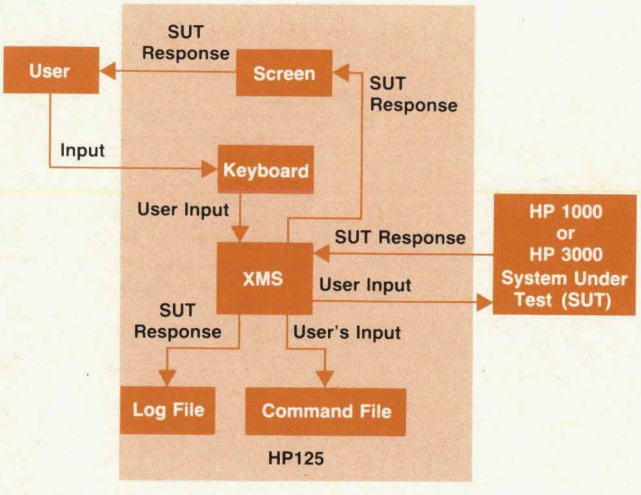


Fig. 1. *XMS data flow.*

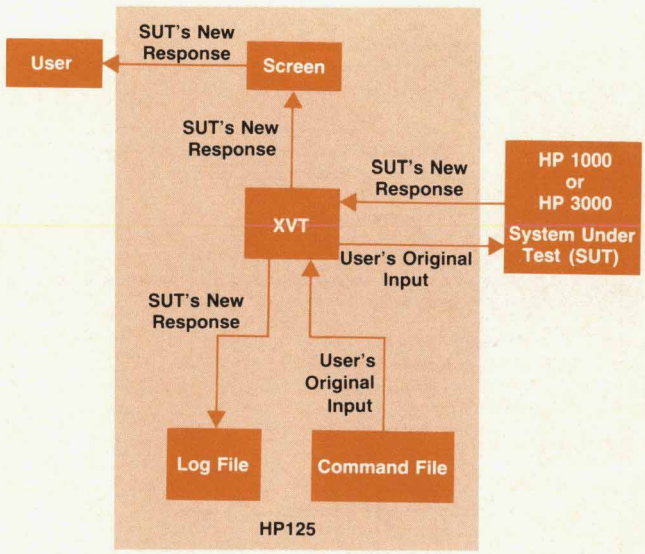


Fig. 2. *XVT data flow.*

typed as if they were entered from a normal terminal. However, all of the activities that occur are recorded (see Fig. 1).

As the test progresses, the engineer manually verifies the responses. In the event that there are no defects, or that any defects that exist are minor enough to allow completion of the tests, the entire set is executed, and the engineer presses the softkey LOCAL OP SYS on the HP 125 to signal completion. In the event that the tests cannot be completed, the software is sent back for correction and testing is begun again with XMS.

The XVT portion of the virtual terminal is generally used when a new software revision is introduced to the test phase, when a new software release (e.g., a product change order) occurs, or when a specific test should be rerun several times until it is passed. When such a need for regression testing arises, the engineer simply reconnects the HP 125 to the system under test, finds the files created by the XMS program, and invokes XVT.

XVT enters the keystrokes typed by the user in the initial run to the system under test. All of the system's responses (including character echoing) are sent both to the screen and to a second log file (see Fig. 2).

XVT and XMS communicate with each other through a command file. All of the user's keystrokes, along with additional information (e.g., some timing data), are stored in this file.

All of the test system's responses are stored in log files on each run, thus creating a history of the system's performance (see Fig. 3).

As stated above, the initial run must be verified manually. However, this can be done after the test has concluded (by listing the log file). Subsequent runs can be verified semiautomatically by comparing the initial log with the current one. Several programs are commercially available for this. We have been using DIFF® by Mark of the Unicorn.

In principle, once a product is ready for release, the tests should be run one final time to obtain a baseline for comparison with any future release.

Besides its use at Data Systems Division for testing HP 1000 operating systems, a special version of VT is being used by HP's Information Networks Division. It uses both the HP 125 and an HP block mode terminal (e.g., the HP 2624) to test forms-mode programs on the HP 3000. Over 1200 test scripts are currently maintained in a library for this tool.

Scaffold Automation Tool and Test Package Standard

The Scaffold Automation Tool and the Test Package Standard were developed jointly by software quality engineering and the HP-UX* validation project. They are currently supported by the Software Engineering Laboratory of HP's Information Technology Group. The HP 1000 version is supported by the Data Systems Division. The motivation behind their development was to provide tools that reduce the time to develop test packages and the time to reuse them in regression testing.

The Test Package Standard defines the physical organization of the test package. This organization is required by the Scaffold and takes advantage of the hierarchical file system. It allows many test suites to be stored in one directory structure and is easily adaptable to varying logical organizations. The Standard also provides skeletons and documentation for writing test plans, test programs, and test package documentation.

The Scaffold provides tools for automating test creation, and for setup, execution, verification, and archival of multiple sets of test results. Verification is done by comparison of the test results with master result files.

The Scaffold requires a physical directory structure as shown in Fig. 4. The directories are divided into two groups: 1) Scaffold tools and administration and 2) the actual test directories.

The contents of the tools and administration directories are:

ADMIN	Scaffold tools
DOC	Scaffold documentation
RESULTS	Record of which tests passed and failed
BAD	Output from tests that failed
GOOD	Output from tests that passed.

Test directories are distinguished by names beginning with lowercase letters.

Each test group directory contains a three-level directory structure as shown in Fig. 5. These three levels are referred to as group-level, section-level, and function-level directories.

The DOC directory should contain the test plan, test pack-

*HP-UX is HP's implementation of AT&T Bell Laboratories UNIX™ System V™ operating system.

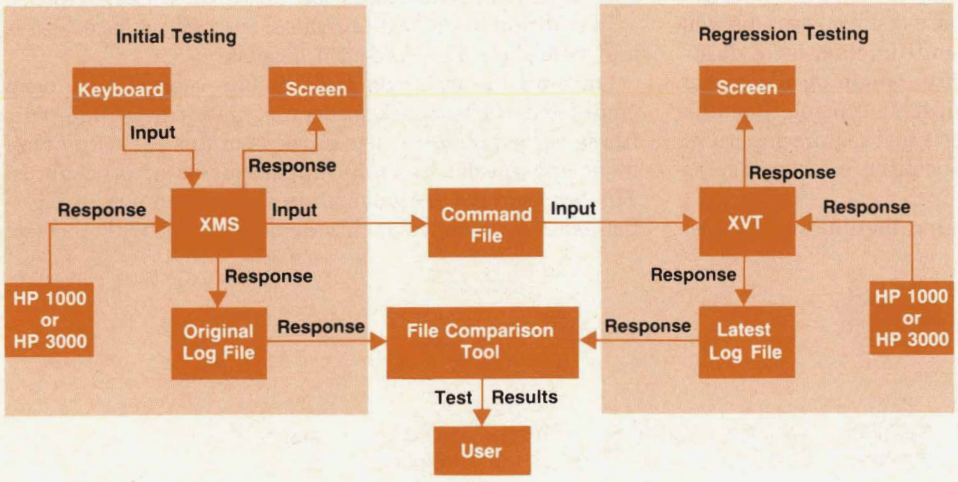


Fig. 3. *Communication between XMS and XVT.*

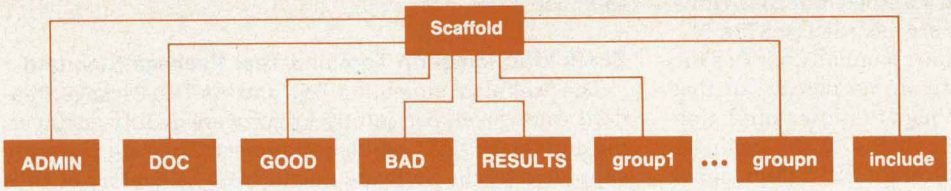


Fig. 4. *Scaffold directory structure.*

age documentation, and test catalog for all the tests in this test section directory. Each function level directory contains all the test programs and scripts for a particular function being tested. The structure of the function level directory is shown in Fig. 6.

The only required file in each function level directory is prog. Optional files are Build, std.out, std.err, std.fil, and std.in. The purpose of each of these files is as follows:

- prog — Executable to run all tests
- Build — Executable to set up test environment
- std.out — Master test output
- std.err — Master error output
- std.fil — Master output to additional file
- std.in — Test input for prog.

include is an optional directory at both the group level and the function level; it may contain files common to many tests at that level.

The group/section/function directory scheme was originally chosen to fit the organization of the *HP-UX Reference Manual*. That is, the group level corresponds to different HP-UX versions, the section level to sections of the *Reference Manual*, and the function level to individual commands and system or library calls.

However, this organization easily adapts to various logical groupings. For example, a group-level directory might contain all the compiler or data base test packages, broken down into appropriate section and function-level directories. Similarly, a section directory containing all tests for a file management package might have separate function directories for interactive tests, tests that require superuser capabilities, tests that require a tape drive, error-producing tests, and other general tests.

The Scaffold tools have an option that allows the user to select any subset of test directories for execution. Thus, with an appropriate breakdown of the test directories, the user can separately run any tests that need special resources. This also allows the user to combine all test packages under a single scaffold structure and selectively run any combination of the test packages.

Test Package Creation

The first step in creating a test package is to write the test plan. This file is named TESTPLAN. The test plan should define all the tests and tools needed, as well as any hardware and software requirements. The test plan skeleton shown in Fig. 7 defines the contents of the test plan. The Standard also provides a skeleton with UNIX nroff document formatting commands.

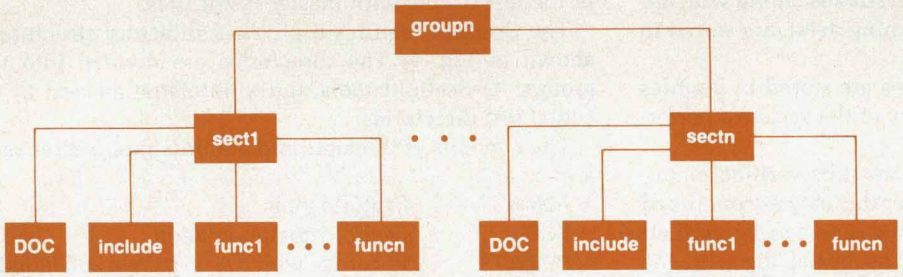


Fig. 5. *Group-level and section-level directory structure.*

The next step is to create the tests and tools defined in the test plan. The Standard defines the header comment section for each test program (see Fig. 8). The create script automates this process. This script creates the file and inserts the header comments, leaving the user in an editor to complete the file.

To insert the header comments, the proper comment character is determined from the file name and inserted around the header. create can also be used to create source files using the UNIX version control utilities SCCS or RCS.

In addition to the test programs, Build, std.out, std.err, std.fil, and std.in should be created if necessary.

Once all the tests and tools in the package have been completed, the test package documentation is written. This file is named README. The purpose of this file is to document any special execution requirements or procedures. The test package documentation skeleton shown in Fig. 9 defines the contents of this document. As with the test

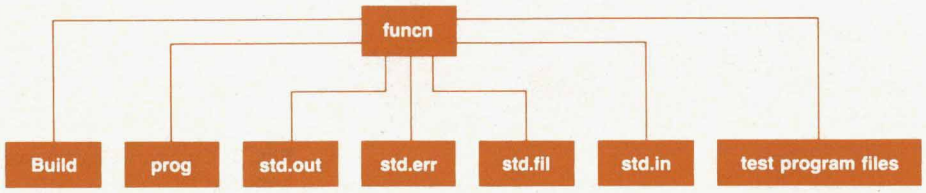


Fig. 6. *Function directory structure.*

1. INTRODUCTION

1.1 Features
1.2 Reference Documents

2. TESTING REQUIREMENTS

2.1 Hardware Requirements
2.2 Software Requirements

3. THE TESTS

3.1 (Test Area 1)
3.2 (Test Area 2)

⋮

3.n (Test Area n)

4. AUTOMATION PLANS FOR THE TEST PACKAGE

4.1 Methods of Automation
4.2 Extent of Automation

5. TOOLS

Fig. 7. *Test plan skeleton.*

plan, a skeleton with UNIX document formatting commands is also provided.

As the tests are being developed, they should be organized and put into a scaffold structure. The tests should be grouped at the section level according to the sections defined in Chapter 3 of the test plan. The TESTPLAN and README files should be put in the section-level DOC directory.

Test Package Execution

The execution of a test package is broken into four parts: setup, execution, verification, and results archival. Each of these steps is automated by the Scaffold tools.

There are two scripts for test setup: gettest and buildtest. gettest is used to create a source scaffold from a scaffold containing version controlled sources in SCCS format, optionally specifying the revision level. The default is to get the latest revision. gettest creates a copy of the SCCS scaffold directory structure, copying directly any non-SCCS files and copying the specified revision of any SCCS files.

buildtest does the actual setup work. Its basic operation is to execute Build in each function-level directory. If Build doesn't exist in a directory, buildtest compiles all C source files in the directory, storing the executable code in prog.

Test execution, verification, and results archival are automated via runtest. runtest executes each prog file in the test function directories, redirecting the standard output and error output to res.out and res.err respectively, and taking input from std.in if it exists.

After prog completes, res.out and res.err are compared to the files std.out and std.err. In addition, if prog created res.fil, it is compared to std.fil. The test is considered to pass if there are no differences, and to fail otherwise. The tests will also pass if std.out or std.err doesn't exist and the corresponding res.out or res.err is empty. Note that although neither std.out, std.err, nor std.fil is required, this is not recommended since it may be impossible to tell if the test passed, such as in the case of a program aborting unexpectedly.

At the start of runtest, a test ID is assigned, which is used to identify the results from this test run, and allows the results of multiple runs of the same tests to be archived. runtest creates a file named test-id in the directory RESULTS. This file contains the names of each function-level test executed and whether each passed or failed.

For tests that fail, all the output is saved in BAD, identified by the test ID and the function-level path name.

No output is saved for tests that pass, unless prog created a directory called savedir in the current directory. savedir is used for output that requires manual verification. If savedir exists, its contents will be copied to GOOD, again identified

The name of this file is ⟨filename⟩

(c) Copyright Hewlett-Packard Company 1985.
All rights reserved. No part of this program
may be photocopied, reproduced or translated to
another program language without the prior
written consent of Hewlett-Packard Company.

Created on ⟨date⟩ by ⟨author's name.⟩

Changes:
⟨include date, name, description and reason.⟩

This file:
⟨One line description of what file tests or does.⟩

Calls tested:
⟨Names of the system calls or commands which are⟩
⟨tested by this program, including the manual ref⟩

Description of this test:
⟨Specify here using as many lines as necessary.⟩

Input parameters:

Expected results:

Side effects of this test:

Supporting files and relationship:

⟨Test-source-code goes here.⟩

Fig. 8. *Test program header skeleton.*

1. INTRODUCTION

1.1 Modification Log
1.2 Reference Documents
1.3 Management Information

2. ORGANIZATION OF TEST PACKAGE

3. DESCRIPTIONS OF TESTS

3.1 General Instructions

3.1.1 H/W & S/W Requirements
3.1.2 Set Up Instructions
3.1.3 Loading
3.1.4 Running
3.1.5 Verifying the Results
3.1.6 What if it doesn't work?
3.1.7 Side Effects

3.2 ⟨Test area n⟩

3.1.1 H/W & S/W Requirements
3.1.2 Set Up Instructions
3.1.3 Loading
3.1.4 Running
3.1.5 Verifying the Results
3.1.6 What if it doesn't work?
3.1.7 Side Effects

⋮

Fig. 9. *Test package documentation skeleton.*

by the test ID and the function-level path name.

There are two ways to specify a subset of tests to buildtest and runtest—either by using a suffix or by specifying the individual directory names.

A suffix has the form <.suffix> and is used to denote a classification of tests that spans many different group, section, or function directories, such as interactive tests or tests requiring superuser capabilities. The suffix must be appended to the names of the files used by buildtest and runtest (Build, prog, std.{out,err,fil,in}, *.c).

If a suffix is given, buildtest and runtest will execute only on directories with files containing that suffix, i.e., Build<.suffix> or prog<.suffix>. If no suffix is given, only files containing no suffix will be used in execution.

In addition, any number of individual directories can be selected. If a group-level or section-level directory is specified, buildtest and runtest will execute on all test subdirectories in each specified directory.

A suffix and directories can both be used. In this case, buildtest and runtest will execute on the designated directories where files with the specified suffix are found.

Acknowledgment

Dave Holt originally designed and developed the Virtual Terminal tool.

Using Quality Metrics for Critical Application Software

Software metrics have been used to evaluate the quality of a computer-based medical device produced by a large-scale software development project.

by William T. Ward

SOFTWARE QUALITY is not a precisely defined parameter. There are several attributes that can be used to measure the quality of software. A list of these attributes might include reliability, maintainability, simplicity of use, testability, understandability of the program code, upgradability, portability, and others.

The nature of the intended application frequently determines how the quality of software will be judged. For example, simplicity of use would probably be a useful software quality metric for a program designed as a word processor for inexperienced typists. Similarly, maintainability of the code might be a useful metric for a large program that is expected to be in use for several years and thus may require modification by programmers not involved in the original design effort.

SQS PROBLEM DESCRIPTION FORM

PRODUCT: _____ QA NUMBER: _____

DATE FOUND: _____ MUST WANT

REPORTED BY: _____

PROBLEM DESCRIPTION:

DATE RESOLVED: _____

DESCRIPTION OF FIX:

Fig. 1. *Standard data entry form completed by the test engineer.*

Reliability of operation is a key aspect of software quality in most applications. The software product discussed in this article is required to provide continuous, accurate monitoring of critically ill patients. The software must be reliable, since unplanned system shutdowns or crashes could jeopardize patient safety.

This article discusses the generation of several software quality metrics from data collected during the system integration stage of the patient monitor software development cycle. The evaluation of these metrics has provided the quantified estimates of software quality required for product release into a critical application environment.

Development Project Overview

The project under study involved the development of the software and firmware for a computer-based ECG monitoring system to be used for continuous cardiac patient surveillance in a hospital coronary care unit.

Approximately four years transpired from the early project design stage to the completion of system testing and resultant release of the product for customer shipment. A total of ten to twelve engineers were involved at some point with the project, and the average staff was seven to eight at any one time. Approximately 85,000 lines of Pascal

source code, excluding comment or blank lines, and 50,000 lines of microprocessor assembler code form the software and firmware basis of the product.

Software Testing Overview

The development sequence for this project closely followed the model as outlined by Fagan.[1] There were a design stage, a coding stage, and a final testing stage. The coding stage included unit and module testing as well as actual implementation. The testing stage of the project included both integration and system testing.

The software testing effort and resultant metrics discussed in this article refer specifically to this final testing phase of the product development cycle. Approximately eight months were required for this effort, using the full-time services of one test engineer and the part-time services of other varied personnel. The total software test effort required approximately 16 person-months.

Black-box functional testing was used extensively in multiple test environments during the software testing stage of this project. The functional testing that was performed can be categorized as either specifications testing, random values testing, or special values testing.

Specifications testing. A free-form prose document, the product External Specification, was created during the design phase of the project. This document was maintained during product development and then passed to the test group at the beginning of the system testing stage.

The External Specification contains the operating specifications for the product software. All functions the product software is designed to perform are listed in the ES. This document formed the basis of the majority of the functional testing performed on the product.

In practice, a checklist of product functionality was generated from the ES and this checklist was used as a test script during system testing.

Random values testing. The intent of this testing technique was to investigate product behavior when the software was exposed to an unrealistic series of input values. Because of the critical nature of the intended product application, the software should respond in a predictable, well-defined fashion, even when exposed to an undisciplined user assault.

An example of a random values test is the input of a series of keystrokes to request a nonconnected sequence of system services. Altered, varying-length keystroke sequences can form the basis for random values testing.

Special values testing. The application of input data that was considered legal but not probable, or that stressed the system, was termed special values testing. The intent of this testing technique was to evaluate the ability of the product code to respond to maximum-configuration conditions or to poorly defined, ambiguous conditions.

An example of special values testing is the evaluation of the product in both the minimum and maximum configurations. These minimum and maximum values can refer to either supported hardware configurations or to various software functions.

Multiple test environments. A major software test methodology used in this project was to create and maintain multiple environments for product testing. Four separate, concurrent in-house test environments and a fully implemented, hospital-based field trial site were used during the software testing stage of this project.

Each of the test environments was used to evaluate the product code under different input conditions. For example, test environment 1 was used for the majority of the specifications testing effort, while test environments 4 and 5 were used for special values testing. Test environment 3 was primarily the random values test station, while the coronary care unit for a hospital served as test environment 2, or the field trial system.

The intent of using multiple test environments is to expose the product to as wide a variety of input conditions as possible. Of primary importance was the feedback provided by the field trial. Failure of a product to perform well in an actual user environment indicates that the design and test efforts preceding the field trial have not been adequate.

Collection and Presentation of Test Data

An Image/1000 data base was created to store information about problems discovered in the product during the software testing effort. The Image utility program KEDIT was used to enter information into the data base. Several Pascal programs were created to retrieve the data from the data base and to present that data in several useful report formats.

Fig. 1 illustrates a standard data entry form. When a

SQS WEEKLY STATUS REPORT

PRODUCT:_____

DATE:_____

SUMMARY OF WEEKLY STATUS CHANGES:

3 items have been reported as NEW.
These are: 255 257 258

4 items have been RESOLVED.
These are: 127 133 249 250

TOTALS FOR ITEMS ON PROBLEM LIST:

3 items are in status NEW.
0 items are in status LAB.
4 items are in status QTST.
12 items are in status NREP.
220 items are in status RES.

Fig. 2. *One of several reports generated from the problem data base.*

RESULTS OF MULTIPLE TEST ENVIRONMENTS

	# OF TEST HOURS	# OF ERRORS FOUND
SYSTEM #1	3840	261
SYSTEM #2	2350	1
SYSTEM #3	1700	10
SYSTEM #4	330	12
SYSTEM #5	330	2

Fig. 3. *Results of the five separate, concurrent test environments.*

problem was found during product testing, the test engineer completed the problem description form. This information was then entered into the data base by means of the KEDIT utility program.

Weekly meetings between the test group and the development group provided a forum for status updates concerning product testing. Fig. 2 illustrates one of the several reports generated from the Image data base by the test group. This particular report indicates which problem items have changed status since the last weekly status meeting. The information in the Image/1000 data base was used to generate each of the software quality metrics.

Software Quality Metrics

Fig. 3 illustrates the results of the five separate, concurrent test environments. Several conclusions can be drawn based on this data:

1. Specifications testing was clearly the most effective testing methodology, yielding an average of 14.73 test hours to find a problem with the code.
2. The field trial results were somewhat surprising, with only a single unique problem discovered after 2350 hours of clinical product use. The implication here is that the in-house test environments together created a superset of the input conditions encountered at the field trial.
3. Although the random values testing was not very productive in terms of errors found per test hour, the problems discovered in this environment tended to be severe, frequently causing system crashes or hangs.

Fig. 4 lists the additional software quality metrics generated for this product at the time of customer release. These values refer specifically to test environment 1, the specifications testing environment. These metrics have been defined as follows:

Test hours logged. 3840 test hours were logged during the software testing effort. This value includes only those times during which active testing was in progress and input data was present for the system.

Count of reported errors. 261 specific problems were found in the product during the software testing effort. Of these reported problems, a full 90% (234) required code changes to fix. The remaining 27 items were classified as not reproducible (13) or as documentation issues needing clarification (12) or as requests for enhancements in a future release of the product (2).

Number of lines of source code. The product software/firmware line count was 135,000. The software consisted of 85,000 noncomment, nonblank lines of Pascal source code. The firmware consisted of 50,000 lines of noncomment microprocessor assembler source code.

Detected errors per 1000 lines of code. With 261 reported errors and 135,000 lines of source code, a ratio of 1.93 errors per 1000 lines of code can be derived. This value compares favorably with typical values reported across the industry for similar software/firmware applications.[2,3,4]

Test hours per error and test hours per 1000 lines of code. 14.73 test hours were required, on the average, to find each of the 261 reported problems. 28.4 test hours, on the average, were spent on each 1000 lines of source code. These values are difficult to interpret, but are useful in planning the amount of time required for software testing of similar products.

Repair time per error. During the software testing phase, each error discovered required approximately 4.3 weeks to repair and retest. The mechanism for reporting newly discovered problems was a weekly status meeting between the test group and the development group, and not all reported problems were solved immediately. These factors tended to inflate this value of 4.3 weeks/error for repair and retest time.

This MTTR (mean time to repair) metric can be beneficially applied to the postrelease life of the product. For example, an estimation of the support group requirements after product release should be influenced by this metric. A high MTTR value might be indicative of a complex product, requiring nontrivial and time-consuming repair efforts. Conversely, a low MTTR could suggest that all the necessary support tools for the product are in place and that future repair efforts will be uncomplicated.

Predicted postrelease field failures. The value of this metric was calculated from the test data using a modified version of a model developed by Simkins.[5] The key inputs to the model are the number and rate of error detection during the testing phase of the project and the estimated effective-

SUMMARY OF SOFTWARE QUALITY METRICS

1) 3840 TEST HOURS LOGGED
2) 261 REPORTED ERRORS
3) 135,000 NONCOMMENT SOURCE LINES OF CODE
4) 1.93 DETECTED ERRORS PER 1000 LINES OF CODE
5) 14.73 TEST HOURS PER DETECTED ERROR
6) 28.4 TEST HOURS PER 1000 LINES OF CODE
7) 4.3 WEEKS REPAIR TIME PER ERROR
8) PREDICTED POSTRELEASE FIELD FAILURES = 1 ERROR PER 6.67 MONTHS

Fig. 4. *Quality metrics generated for the patient monitor software product at the time of customer release.*

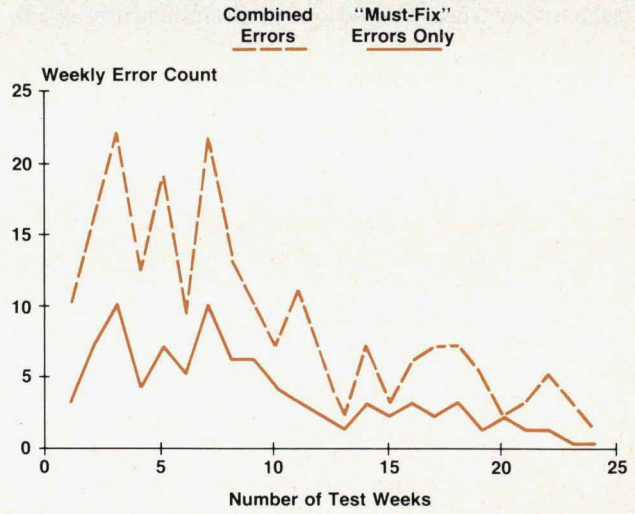


Fig. 5. *Rate of error detection during the software testing phase.*

ness of the test environment relative to the field environment. The predicted value for this metric of 1 error per 6.67 months of field exposure refers specifically to problems of a serious nature that would impair the proper functioning of the system.

Rate of error detection. Fig. 5 illustrates the rate of error detection experienced during the software testing phase. Assuming a fairly constant testing effort, the graph of Fig. 5 indicates that further testing of the software in an unchanged environment would probably yield little productive return.

Conclusions

Various techniques and metrics can be employed to assure and quantify the high level of quality required of critical application software before release to customer use. Several conclusions concerning these techniques and metrics can be drawn, based on the product discussed in this article.

Specifications testing can be very productive in terms of problems discovered per test hour. The key requirement for the successful application of specifications testing is an accurate statement of product functionality—a document similar to the External Specification discussed in this paper. This document can be the basis for the specifications test script.

Multiple test environments can be used to good advantage during software testing activities. The greater the diversity of the input data applied to a product under test, the higher the probability that problem areas of the code will be discovered. Another potential advantage of multiple test environments is that the use of different hardware sets to coincide with each test environment can highlight possible hardware/software interaction problems not readily apparent with a single hardware test set.

It is particularly important that the product be exposed to a environment that closely approximates the intended use environment before release of the product. A properly implemented field trial is one method for achieving this type of product exposure. Substantial testing must be performed on the product before its introduction to a live field trial environment. This is necessary to ensure that unexpected, severe problems in the code at the field trial do not jeopardize the critical-application use of the product.

The release of software destined for a critical application should be contingent on product metrics that satisfy a precisely defined range of values. For example, one of the strictures for product release might be the requirement that a specific number of continuous test hours elapse without the discovery of a problem in the code. One of the requirements for the product discussed in this paper was a week-long interval of test time without the appearance of any serious problems.

A logical expression that relates the various release requirements for a software product based on metrics might be as follows:

Product Release State := S1 AND S2 AND S3 AND S4 AND S5

where S1 through S5 represent test state conditions defined as:

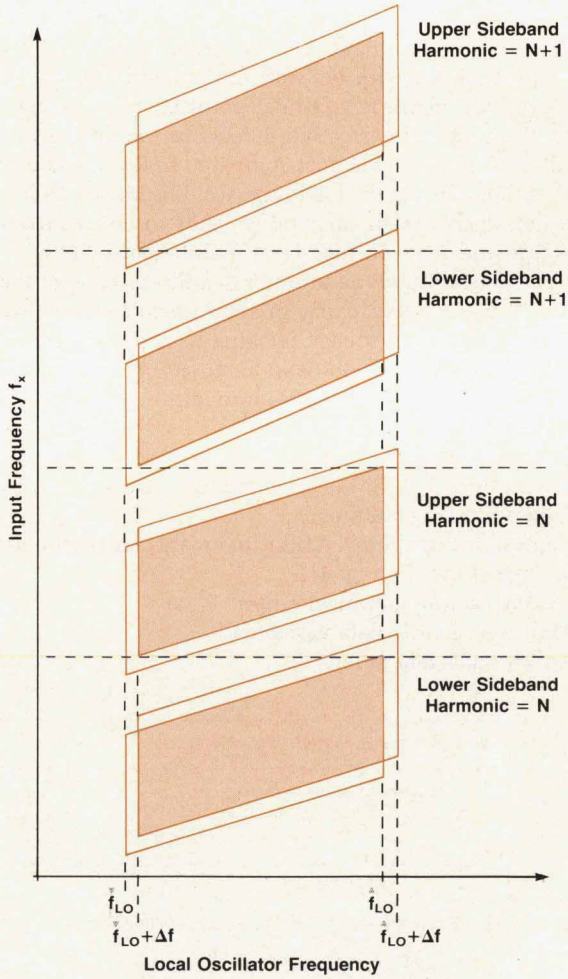
S1 = Test hours >25 per 1000 lines of code
S2 = Errors per 1000 lines of code in range of 0.5 to 10
S3 = Rate of error detection decreasing for two weeks
S4 = No serious errors found in last week of testing
S5 = Field trial hours >½ total in-house test hours.

References

1. M.E. Fagan, "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, Vol. 15, no. 3, 1976, pp. 182-211.
2. V.R. Basili and B.T. Perricone, "Software Errors and Complexity: An Empirical Investigation," *Communications of the ACM*, Vol. 27, no. 1, January 1984.
3. P.N. Misra, "Software Reliability Analysis," *IBM Systems Journal*, Vol. 22, no. 3, 1983, pp. 262-270.
4. M.L. Shooman, *Software Engineering Design/Reliability Management*, McGraw-Hill, 1983, pp. 226, 326, 350-384, and 432.
5. D.J. Simkins, "Software Performance Modeling and Management," *IEEE Transactions on Reliability*, Vol. R-32, no. 3, August 1983, pp. 293-297.

CORRECTION

In the February 1986 issue, the shaded area in Fig. 3 on page 12 was shown incorrectly. The correct figure is shown below.



P-PODS: A Software Graphical Design Tool

P-PODS enforces formal software design, allows designs to be maintained on-line, and produces output suitable for design walkthroughs.

by Robert W. Dea and Vincent J. D'Angelo

P-PODS (Pictorial Procedure Oriented Design System) is an interactive graphical software design and documentation tool. Available for internal Hewlett-Packard users only, its target users are software R&D engineers. As a design tool, P-PODS is used during the design phase of a project to replace the pseudocoding or flowcharting of detailed logic structure that would normally be done. The resulting diagrams supplement information available in the finished code. As a documentation tool, P-PODS is used to document existing code.

With P-PODS, an engineer interactively creates a design diagram. These diagrams are kept on-line and are easily changed. Hard-copy output of these diagrams can be produced for formal design reviews or for internal maintenance purposes. Once the designs for the project have been finalized, code templates can be generated to assist the software engineer with the start of the coding phase.

The design phase is a critical point for eliminating defects. Finding and correcting defects in the design phase results in a much lower cost compared to finding and correcting them during the testing phase. Finding elements of complex design at an early point leads to better program structure and reduced long-term maintenance costs.

P-PODS is a prototype product to address some of these issues. It was created partly to obtain feedback for future design tools. This feedback is being incorporated into future R&D efforts in the Software Engineering Laboratory of HP's Corporate Engineering Department.

Phased Release

Originally, P-PODS was intended to be a design tool that had the following features:

- Hierarchical graphical editor (to create charts similar to the one shown in Fig. 4).
- Logic structure graphical editor
- Ability to handle data variables
- Partial code generation

```
PROCEDURE:  count_options        Counts students in CS
WHILE  student records remain to process
    retrieve student record
    IF  computer science major
        THEN
            CASE  CS option
                systems
                    add 1 to systems count
                math
                    add 1 to math count
                applications
                    add 1 to application count
        ELSE
            add 1 to non-CS count
```

Fig. 2. *Example of a P-PODS design equivalent to Fig. 1.*

- Code scanners to bring existing code into the design system
- Limited design analysis.
- Calculation of complexity metrics.

It was decided that a phased release approach was appropriate to get important user feedback. The project was partitioned into six phases. The first phase developed the logic structure graphical editor on the HP 3000 Computer. This was the minimum core subset of functionality. The second phase enhanced this version, adding partial code generation capability. The third phase was a port of the phase two release to the HP-UX operating system, with automatic calculation of design complexity. The proposed fourth through sixth phases were to add the remaining features (the hierarchical editor, etc.) to the HP-UX version.

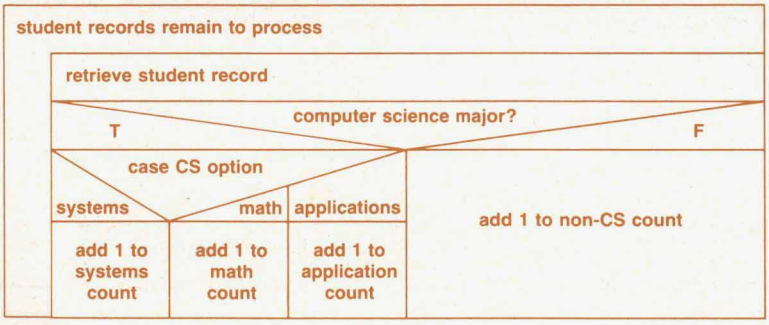


Fig. 1. *A Nassi-Shneiderman design example.*

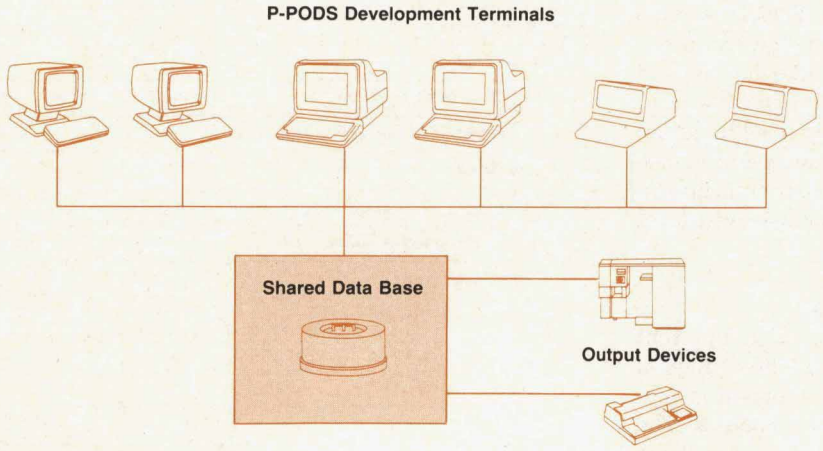


Fig. 3. *The P-PODS/3000 environment.*

P-PODS Diagrams and Features

P-PODS provides an interactive tool for structured flowcharting. The design diagrams created are combinations of individual design constructs. These design constructs are similar to Nassi-Shneiderman structure flowcharting constructs.[1] An example of a Nassi-Shneiderman diagram is shown in Fig. 1.

Although P-PODS design constructs are based on the Nassi-Shneiderman representations, alterations were made to avoid the subdividing of the diagram into narrow vertical columns for decision representations. These alterations allow the terminal's screen space to be used more effectively. Fig. 2 shows the P-PODS equivalent of the example shown in Fig. 1.

P-PODS provides the following structured design constructs: SIMPLE, WHILE, FOR, PROCEDURE CALL, IF, ELSE, CASE, CASE ELEMENT, DO UNTIL.

P-PODS prevents the user from producing incorrect design structures. It rejects any attempt to combine sequences of incompatible structures. For example, the user would not be allowed to add a second ELSE construct to an IF construct.

Multiuser Environment

P-PODS is designed to be used by a project team. Each project member's designs are stored in a single shared P-PODS data base containing all of the designs for the whole project. Each person on the team can interrogate the data base at any time to view its contents. Fig. 3 shows the P-PODS environment on the HP 3000 Computer.

The current underlying design data base is an Image data base, and all Image-related support tools are available for maintaining it.

In P-PODS, designs within a project are partitioned into subgroups called modules. Modules are logical groupings of P-PODS design diagrams. For example, a project team might decide to define a module to contain graphics related routines, while another module might contain user interface related routines, and so on. A module in P-PODS is a lockable entity, preventing two project members from changing the same design at the same time.

P-PODS can translate its designs into either Pascal or C code templates. The user can then edit the generated file with any favorite editor to fill in the missing pieces. This feature relieves the user of the need to match BEGIN/ENDs, etc. P-PODS also translates the design construct descriptions into comments, and places these comments in the appropriate places in the generated code template.

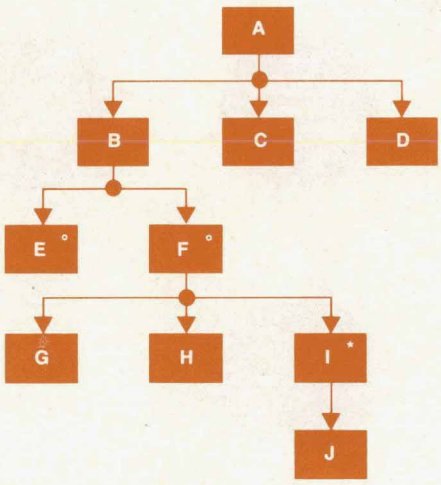


Fig. 4. *Example of a Jackson diagram.*

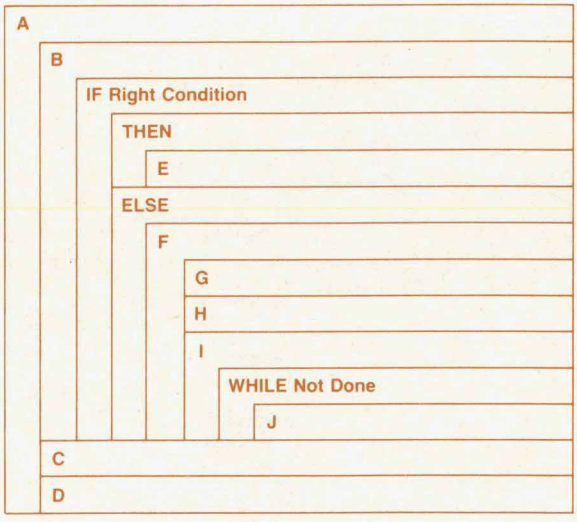


Fig. 5. *Example of P-PODS constructs to represent hierarchical structure, equivalent to Fig. 4.*

Several copies of P-PODS/3000 and P-PODS/UX have been distributed to software engineers in the company. Feedback has indicated that users would like design tools containing interactive hierarchical and data flow diagramming capabilities. Future design tools also need to be implemented on workstations to overcome any performance problems caused by host-to-terminal communications. Also, some users prefer an enhanced editor to a lower-level design tool.

Future Uses for P-PODS-Type Diagrams

P-PODS design constructs can also be used to represent hierarchical structures. These constructs can show program modules and their interrelationships. A program is represented as a hierarchically ordered set of modules. The modules that perform the higher-level functions are closer to the left margin. Lower-level modules are indented, and are lower in the diagram than their parent modules.

For example, the Jackson diagram[2] shown in Fig. 4 has the equivalent representation using P-PODS-type constructs shown in Fig. 5.

P-PODS-type design constructs can also be used to represent hierarchical data structures. This representation shows a program-level view of the data. For example, a Warnier-Orr diagram[3] is shown in Fig. 6, and its equivalent P-PODS-type representation is shown in Fig. 7.

Using the hierarchical data structure and detailed design representations discussed so far, a method of design can be derived that integrates these design techniques. Software engineers can use any subset or all three of these design representations in designing software. For example, the software can first be designed in a hierarchical manner. From there, detailed design can be performed for any of the components in the hierarchical design. As the detailed design is progressing, data structures can be defined using the data structure representation. This type of environment is shown in Fig. 8. A transfer from one design representation to another exits back along the same path from which it was invoked. In this way, three different kinds of design representations using P-PODS-type constructs can be used in the design of software.

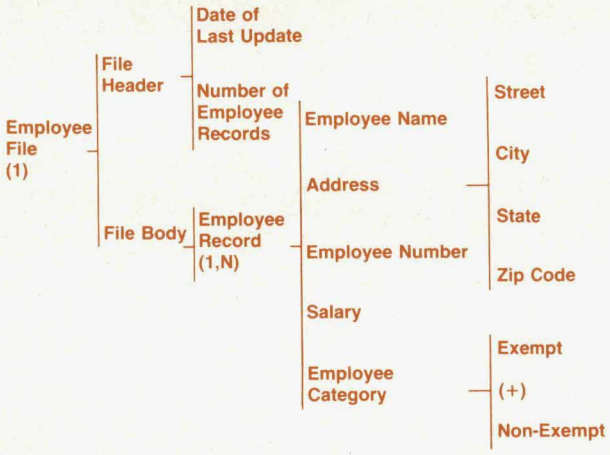


Fig. 7. *Example of P-PODS constructs to represent data structure, equivalent to Fig. 6.*

Conclusion

The design phase is a critical point of a project for preventing defects. Defects introduced during design increase development costs, extend development schedules, and frequently create large maintenance costs. The techniques available through P-PODS and other tools currently under exploration deal with some of the primary causes of defects, and will lead to better software products.

Acknowledgments

We would like to give special acknowledgments to Debbie Caswell for her contributions to P-PODS/UX, and to Bob Grady for giving us the opportunity to develop P-PODS.

Fig. 6. *Example of a Warnier-Orr diagram.*

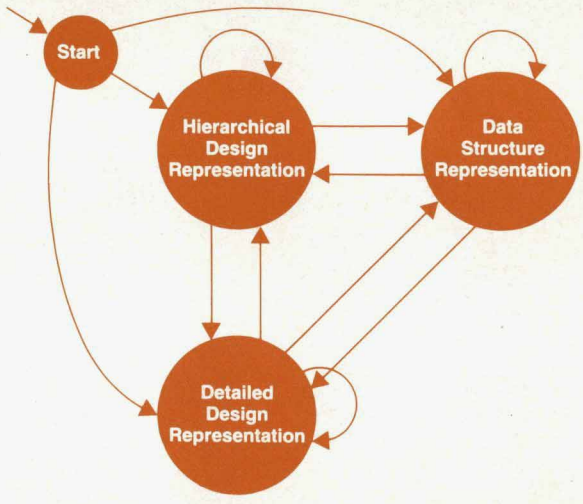


Fig. 8. *Example of future design environments.*

References

1. I. Nassi and B. Shneiderman, "Flowchart Techniques for Structured Programming," *ACM SIGPLAN Notices*, Vol. 8, no. 8, August 1973, pp. 12-26.
2. M.A. Jackson, *Principles of Program Design*, Academic Press, 1975.
3. J. Warnier, *Logical Construction of Systems*, Van Nostrand Reinhold Company, 1981, pp. 11-38.

Triggers: A Software Testing Tool

Triggers as a software testing method focuses on testing the boundary conditions of the software, and enables the software tester to be more productive.

by John R. Bugarin

SOFTWARE TESTING is in the eyes of the world a black art. This art contributes to the quality of the software product and consumes a large amount of effort in the software development life cycle.

Triggers is a software testing method to increase the productivity (efficiency and effectiveness) of testing. It allows the tester to force the execution of specific paths in the software by setting specific software conditions.

Software projects consist of several different partitions called modules. Consider the number of execution paths between these modules. If we assume ten modules, each having only one interface, then there are potentially 72 different intermodule entry point paths. However, this number includes only the number of module entry point pairs, it does not include different combinations of module entry point sequences. In most software projects, the number of different combinations is quite large.

How can the writer of module X test X's interfaces with modules A, B, C, and D? How can the writer of module X increase the testing branch flow coverage of X? Triggers is the answer.

This method can easily be implemented in most development languages. However, I will not address implementation here. Like all methods, Triggers is better explained through an example, and I will use the language MODCAL for the Triggers example that follows.

MODCAL Example

This Triggers implementation is based on the exception handling (TRY/RECOVER and ESCAPE) mechanism of MODCAL. The grammar of the MODCAL TRY/RECOVER and ESCAPE statements is:

```
⟨statement⟩ ::= TRY ⟨statement list⟩ RECOVER ⟨statement⟩
            ::= ESCAPE ( ⟨expression⟩ )
```

The ESCAPE statement's expression is integer-valued. If during the execution of the ⟨statement list⟩ in the TRY/RECOVER statement an ESCAPE statement is executed, the program will continue execution at the ⟨statement⟩ following the RECOVER. Hence, several levels of procedure calls could potentially be aborted. Consider the following example:

```
                PROCEDURE X;
                  BEGIN
                      ...
statement 1:          TRY
statement 2:              Y;
statement 3:          RECOVER
statement 4:              WRITELN("BINGO!");
                      WRITELN("BANGO!");
                       ...
                  END;(*X*)

                PROCEDURE Y;
                  BEGIN
statement 1:          Z;
                  END;(*Y*)

                PROCEDURE Z;
                  BEGIN
statement 1:        IF (⟨an error condition⟩) THEN
statement 2:          ESCAPE (⟨the error condition⟩);
                    ...
                  END;(*Z*)
```

Upon entry of procedure X, the TRY/RECOVER statement is entered (statement 1 of X). Procedure Y is called and immediately procedure Z is called. At statement 1 in procedure Z an error condition is encountered and an ESCAPE statement is executed (statement 2 of Z). The run-time system then searches for the innermost TRY/RECOVER statement (procedure X in this example) and executes the recover statement (statement 4 of X). "BINGO!" is printed. Execution continues after the TRY/RECOVER statement and "BANGO!" is printed.

The above example is simplified because no recursion of the TRY/RECOVER statement exists and no ESCAPE statements are executed in the recover statement. Both TRY/RECOVER and ESCAPE recursion are supported in MODCAL.

Using the example on the preceding page, add a compiler directive and a new procedure call Try_Trigger at the beginning of procedure Z:

```
                PROCEDURE Z;
                  BEGIN
                    $IF TRIGGER_ON$
                      Try_Trigger (...);
                    $END$
statement 1:          IF ((an error condition)) THEN
statement 2:            ESCAPE ((the error condition));
                      ...
                  END; (* Z *)
```

Upon entry of Z, Try_Trigger is called. It queries a trigger data base. If an entry is found, Try_Trigger executes an ESCAPE statement with the designated escape value. Otherwise, Try_Trigger is a NO OP. With this mechanism a condition is triggered; hence the name Triggers.

An instance of a trigger consists of two search keys (a module name and a trigger number), a reference count which is decremented when the trigger is called by Try_Trigger, and an escape value which is used when the reference count is equal to or less than zero.

All interfaces between modules in the project can be identified and assigned trigger numbers which can made public in a MODCAL definition file. These numbers are negative values to allow individual testers to use positive numbers for their own triggers. At every interface, the trigger is placed as follows:

```
PROCEDURE XXX (...);
  BEGIN
    $IF TRIGGER_ON$
      Try_Trigger (My_Modname, T_XXX);
      Try_Trigger (ANYBODY, T_XXX);
    $END$
      ...
  END; (* PROCEDURE XXX *)
```

My_Modname is a function that returns the identifier of the current module and T_XXX is a constant. ANYBODY is a constant modname that denotes any module. The first Try_Trigger call provides a testing interface. Any module can use this interface to test specific interfaces to procedure XXX. The second Try_Trigger call provides the ability to test the general interface to procedure XXX.

With triggers in place at every interface, the writer of module X can test the interfaces to all client modules. The module writer identifies the sequence of intermodule interactions that need to be tested and uses triggers to force the chosen execution paths. This same concept can be applied to the internal interfaces of the module (intramodule). Furthermore, triggers can be used to increase branch/path flow coverage.

The implementation of Triggers consists of the data base (with insertion and deletion operations) of records consisting of the two keys (module name and trigger number), the reference count, and the escape value. Interactive and programmatic interfaces for inserting and deleting triggers in the data base should be implemented.

Triggers can be applied to several other testing needs by using the general model below:

```
$IF TRIGGER_ON$
TRY
    Try_Trigger (My_Modname, TRIGGER_NUMBER);
RECOVER
    ⟨special software⟩
$END$
```

If Try_Trigger escapes, the special software will be executed. Otherwise, the code is equivalent to a NO OP. This model can be applied to test module interfaces that consist of more than just escapes by executing special software to trigger the software condition needed.

Problems

This trigger example exposes several problems. The Try_Trigger procedure calls are manually identified and inserted into the software. Hence, identifying and inserting all the triggers required is a painful process. The module name concept needs to be implemented across the total software system. Resolution of a trigger in this example is at the module level and not at the process level. Hence, only one trigger test per module can be executing at a time.

Triggers Experience

The key question is "How productive (efficient/effective) is Triggers?" We have used this MODCAL version of Triggers on a project of 15 modules written by 12 engineers. We asked the engineers what percentage of defects found were found using Triggers and their answers ranged from 40 to 80%. Our personal experience was around 95%, probably because of our intimate knowledge of Triggers. During the last month before manufacturing release, 31 major defects were found in the first week using specific and general triggers (an investment of one engineer week).

Even without quantitative data, our organization has accepted Triggers as an effective testing tool to be used on future projects.

Acknowledgments

Triggers was developed during the Hammer Project (LAN/500) and several people deserve recognition for their roles. The project members were Tim DeLeon, Bill Mowson, Mike Robinson, Mike Shipley, Charlie Solomon, Dean Thompson, and Mike Wenzel. I am grateful for their patience during the evolution of the definition of Triggers. Special thanks to Carl Dierschow for his active role in Triggers and to the project managers, John Balza and Jim Willits, for providing the support and environment to allow the creation of Triggers.

Hierarchy Chart Language Aids Software Development

HCL is used by software designers at several Hewlett-Packard Divisions to speed up the process of generating hierarchy charts.

by Bruce A. Thompson and David J. Ellis

TODAY, SOFTWARE DESIGNERS are using structured methods that organize the development process from specification through final code into a sequence of steps. The end of each step is marked by the creation of part of the overall documentation. This documentation greatly facilitates communication between designers and improves the maintainability of the product.

One of these pieces of documentation is a hierarchy chart, which is a graphical representation of the structure of the software. It depicts the organization of the modules. (A module is a simple procedure, function, subroutine, or similar atomic piece of code.) With this chart, engineers can analyze the proposed design with a view to making improvements.

A hierarchy chart allows careful examination of the program's binding and coupling before code is written. Binding is a measure of the functionality of a module. At the top of the binding scale are modules that do one and only one task, while modules made of random blocks of code are at the bottom of the scale. Coupling is a measure of the traffic between modules. A module that modifies the code of another has high coupling, whereas two modules that pass little or no data have low coupling. A further discussion of coupling and binding can be found in references 1, 2, and 3.

This is where the designer can make the best effort towards ensuring program modularity and future reusability. A program that is designed with high binding and low coupling will be much easier to modify or repair. It will also be much easier to reuse modules designed in this way.

Before this analysis can be performed, however, the chart must be drawn.

Problems with Existing Methods

Traditionally, the software engineer draws hierarchy charts in one of two ways: by hand or with a generic graphics editor. There are problems with both methods. Charts drawn by hand vary widely in style and are very time-consuming to produce. General-purpose graphics editors, although powerful, can be difficult to learn and use. What these editors lack is the specific knowledge of hierarchy charts needed so that changes can be made quickly and easily. Both methods continually confront the engineer with the topology problem—laying out the chart.

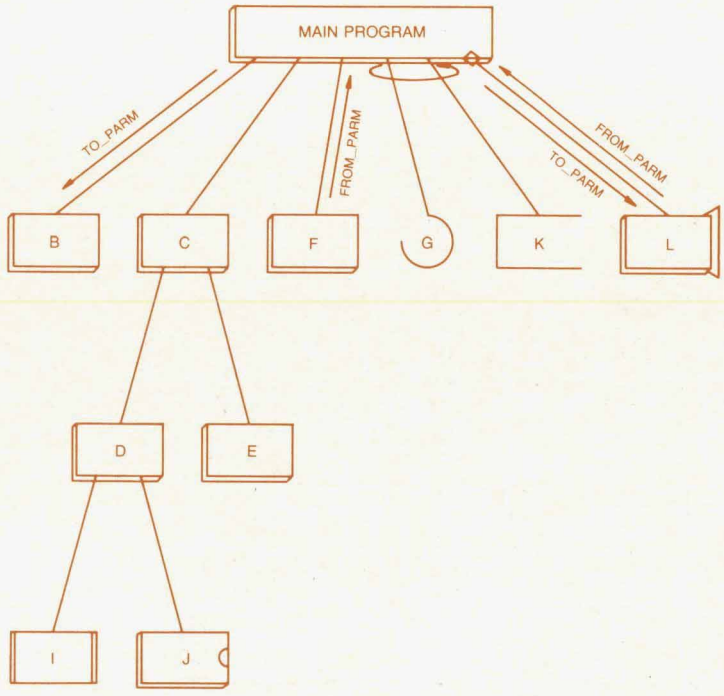


Fig. 1. *An example of a simple hierarchy chart showing the different types of modules.* MAIN PROGRAM,B,C,D,E,F, *and* H *(not called) are* modules. G *is a* system *module.* I *is an* external *module.* J *is a* hardware *module.* K *is a* data *module.* L *is a* recursive *module. Module names can be up to 32 characters long. HCL draws each module name on up to 3 lines within a symbol.*

As the number of modules on the chart increases, there is an exponential increase in the difficulty of this problem. The engineer must place the modules to make the chart clear and understandable. Even minor modifications to the chart can require starting over and redrawing the chart. The time required for redrawing charts can make the designers feel that they are spending more time drawing charts than designing, and tends to make the designers reluctant to make modifications. This reluctance has often caused designers to defend their original designs rather than admit that they could be improved.

The solution to these problems is a graphics program, aimed specifically at generating hierarchy charts, that requires little time to learn or operate. The Hierarchy Chart Language (HCL) program is primarily a software engineering tool used within several HP Divisions. It was developed to facilitate the use of structured software design. HCL grew out of the need to generate hierarchy charts quickly and easily so the designers could concentrate on the design rather than the representation of software. HCL automatically places modules and routes interconnections. This is the most time-consuming aspect of chart generation for the engineer to do by hand. This 100% placement and routing is not restrictive, however. The designer is still allowed the flexibility to alter the appearance of the hierarchy chart to conform to a personal style.

In designing HCL, there were two choices for the input: a text file or interactive graphics. The text file input was chosen, mainly because it could look like a block-structured language, something that most software engineers are very familiar with. Also, if an engineer wants to make a hierarchy chart for an existing piece of code, it is easier to do if the input to HCL is text. To provide text input, the user does not have to learn yet another editor, but simply uses any familiar one. By building on the knowledge base the software engineer probably already has (i.e., text editors, block-structured languages, hierarchy charts), HCL requires the engineer to learn only its very simple syntax.

Language Format

The primary goal in specifying the hierarchy chart language was to minimize the information needed to draw a hierarchy chart. This can be summarized as a list of the modules that appear on the chart and how these modules fit together. This list is the basis for the overall language structure.

The input is divided into three sections: options, declarations, and definitions. Most compilers of high-level languages divide the input into similar sections, so again, this is something that the user is familiar with.

The options section gives the engineer control over the appearance of the hierarchy chart. There are options to label the chart with a title, the author name(s), the date, and a legend or key. The user can disable certain features of HCL, such as the drawing of parameters and parameter checking. In addition, this section provides control over the operation of the program.

Before a module can be used in the definition section, it must first be declared. This is similar to the type declaration of variables in many of the high-level languages. Types and modules can be declared in any order.

HCL provides support for many different module types. Each type is drawn differently on the chart and has different uses.

The first module type is the simple *module*, which represents common procedures, functions, subroutines, etc. The *recursive* module is used to represent modules that call themselves, either directly or indirectly. These modules are generally more complex than the simple modules and are therefore shown differently. The *external* module is used to represent modules that are not defined on the chart. These modules may be defined on another chart, or

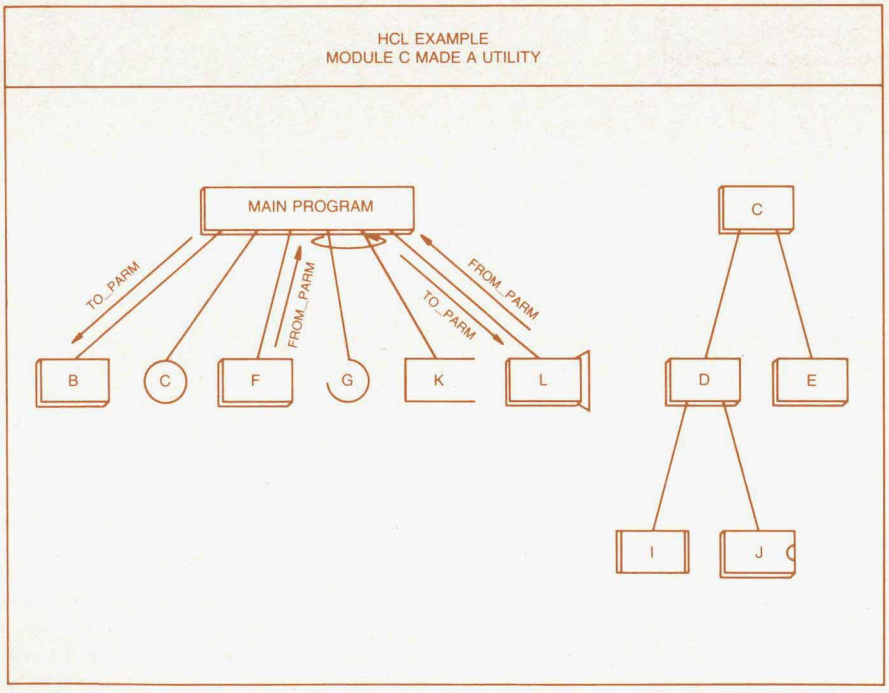


Fig. 2. *A hierarchy chart in which module C is drawn as a utility module.*

may remain to be designed. An external module is one way to show possible enhancements to a program on the chart.

The *system* module is used to represent operating system calls, such as modules that read the system clock or open a file. The *data* module was devised to show access to data such as variables, data bases, files, etc. During design, it is important to show which modules access the data so that the interactions (coupling) of that data can be minimized.

The *hardware* module was added specifically to show software interfaces to hardware registers. This allows the assessment of coupling of not only the software-software interface but the hardware-software interface as well. A hardware module looks like the integrated circuit representation used on schematic diagrams by digital designers. Hardware interactions with software are especially important in microprocessor-based control software.

The *invisible* module is shown by simply drawing the name of the module. This module can be used as a generic module type, making HCL useful for applications other than software hierarchy charts. Fig. 1 is an example of a simple chart showing the various module types.

The limit on the length of a module name is 32 characters. A name can contain nearly any printable character. To make the module names easier to read, HCL draws the name on up to three lines within the module shape. The underbar character is used as a place for HCL to divide the name into multiple lines if the name is too long. The underbars are not drawn on the chart.

The definition section is where the structure of the hierarchy chart is specified. Modules are defined to call other modules by listing subordinate modules in a Pascal-like begin-end block. Parameters can be included each time a module is used. These parameters are divided into two types: those passed to a module and those returned from the module. When using a module more than once, the number of to and from parameters must be the same. However, the parameter names do not have to match. Nearly all printable characters can be used as parameters. This allows the user to customize representations for the parameters such as separating control and data parameters. For example, the data parameters might be contained in brackets and the control parameters simply listed.

Conditional calls, case statements, and looping constructs are supported. Each of these can have a begin-end block to represent that more than one module is affected.

Nesting of loops can be done to show nested iterations on the hierarchy chart. However, trying to nest case or conditional calls will result in an error. This is because there is no clear way to show this type of nesting on a chart.

The following code is the very simple textual definition that produced the chart shown in Fig. 1.

```
MODULE MAIN PROGRAM,B,C,D,E,F,H;
SYSTEM G;
EXTERNAL I;
HARDWARE J;
DATA K;
RECURSIVE L;
MAIN PROGRAM
  BEGIN
  B(TO_PARM);
  C
   BEGIN
   D
    BEGIN
    I;
    J;
    END;
   E;
   END;
  F(/FROM_PARM);
  *LOOP
   BEGIN
   G;
   K;
   END;
  *COND L(TO_PARM/FROM_PARM);
  END;
```

A problem arises in drawing a chart when a module is

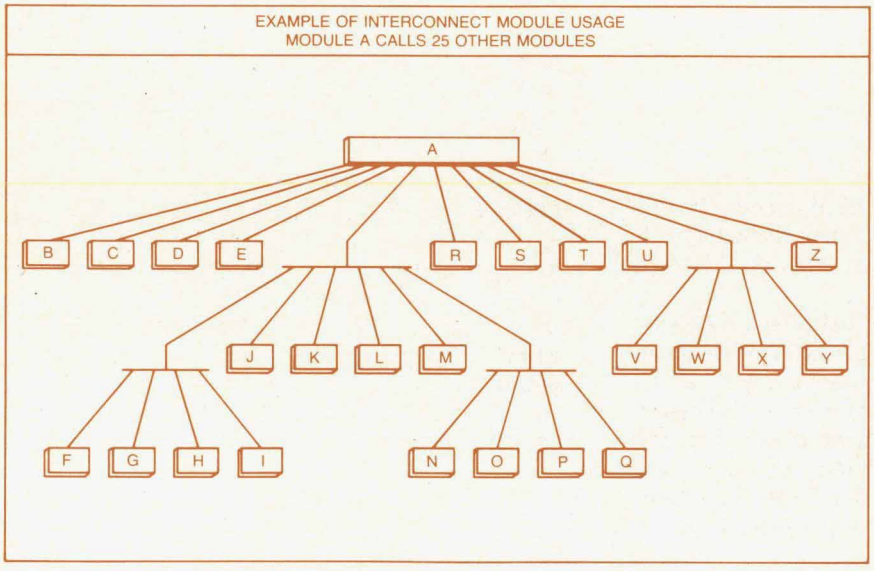


Fig. 3. *An example of the use of the interconnect module. Module A calls 25 other modules. HCL allows up to 512 modules per sheet of paper. Programs that have more than 512 modules can be drawn on two or more sheets.*

called more than once. One approach is to use a circle with a unique letter inside. Whenever the module is called, it is replaced with its corresponding letter in a circle. The module is then drawn off by itself with any module calls it makes drawn underneath.

For a large chart it can become difficult to remember which letter represents which module. Another approach is to draw the module near the bottom of the paper and draw lines to this module for every call. This results in a chart of many lines and few modules.

When HCL detects a module's being called more than once, it replaces the call with a circle but inserts the actual name of the module in the circle. The circle is called a *utility* module. The module is then drawn by itself as a subchart along with any modules it may call. In this way, the structure can be understood easily and the problem of many lines bisecting the drawing is avoided. The following code provides an example.

```
MODULE MAIN PROGRAM,B,C,D,E,F,H;
SYSTEM G;
EXTERNAL I;
HARDWARE J;
DATA K;
RECURSIVE L;
MAIN PROGRAM
  BEGIN
  B(TO_PARM);
  C
   BEGIN
   D
    BEGIN
    I;
    J;
    END;
   E;
   END;
  F(/FROM_PARM);
  *LOOP
   BEGIN
   G;
   K;
   END;
  *COND L(TO_PARM/FROM_PARM);
  END;

C;     {MAKES MODULE C A UTILITY}
```

Fig. 2 shows the chart produced by this code. In this example module C was made a utility module by listing it again at the end of the file as shown. Module C is then drawn separately as a subchart.

One problem that can occur when defining a hierarchy chart is a module that eventually calls itself. This is known as recursion. Sometimes this is desirable, but it can be disastrous if unintentional. There are two types of recursion that HCL checks for, direct and indirect. Direct recursion is a module calling itself from itself. Indirect recursion is a module calling itself through intermediate modules. Indirect recursion is the most difficult type of recursion to detect manually.

When either type of recursion is identified and the module has not been previously declared a recursive type, a warning is generated and the module type is changed to recursive.

It is often necessary to define separate subcharts on a single drawing. This is useful for showing functional partitioning of the design as well as concurrent processing. The user may achieve this effect by simply including the definitions of these charts separately, one after another in the file. HCL will draw each as a separate subchart on the drawing.

A common nuisance in many block-structured languages is the requirement that a symbol be completely defined before it is used. In Pascal a procedure name must be fully coded before it can be called. HCL allows a module to be called anywhere in the file without reference to where the module is defined. This permits easy reordering of the text within the file.

A common cause of problems for hierarchy charts is a module that calls a sequence of many modules. HCL will draw the module calls all on the same level, creating a very wide and short chart. The *interconnect* module can be used to draw some of these modules at a lower level on the chart and still maintain the structure. The interconnect module was added to allow the user to make the chart easier to read; it is ignored when the cross references are generated. The following code and the resulting chart (Fig. 3) show an example of the use of the interconnect module.

```
*TITLE 'EXAMPLE OF INTERCONNECT MODULE USAGE';
*NAME 'MODULE A CALLS 25 OTHER MODULES';
*MODULE_DEFAULT;
*NO_WARNING;
INTERCONNECT LINK1,LINK2,LINK3,LINK4;
A
 BEGIN
  B;
  C;
  D;
  E;
  LINK4
  BEGIN
   LINK1
   BEGIN
    F;
    G;
    H;
    I;
   END;
   J;
   K;
   L;
   M;
   LINK2
   BEGIN
    N;
    O;
    P;
    Q;
   END;
  END;
```

```
  R;
  S;
  T;
  U;
  LINK3
  BEGIN
    V;
    W;
    X;
    Y;
  END;
  Z;
END;
```

The Topological Problem

The real power of HCL is its ability to draw any chart specified by the language automatically. The user does not have to perform any graphical operations to generate a chart. The process HCL uses to draw a chart from the text supplied consists of three steps.

The first step is to make a first-pass computation of the chart. This step makes a rough estimate on how the modules should be placed. It does not try to put the boxes as close together as possible nor does it attempt to center the boxes. It does, however, place the boxes and circles so that they do not overlap and the associated parameters do not cross.

The second step, compacting, takes all the subcharts and moves the boxes and circles as close together as possible. Each module is centered above those that it calls to make the chart look better.

The third step is to take all of the subcharts and organize them to fit a specific paper size. All of the subcharts are arranged in one long row, which is then chopped into pieces and the pieces arranged to fit the length-width ratio of the paper. The number of subcharts in each row depends upon the size of the subcharts. The subcharts are ordered from left to right and top to bottom, starting with the subchart first defined in the input text. The order then follows the sequence of the modules in the input text. Any utility modules are located after all the subcharts.

After this three-step process is performed, the chart is ready to be drawn. However, this does not limit modifying the arrangement of the chart. There are several ways to change the appearance of the chart. One method is to change the order of the subcharts in the input file. This will change the order of the subcharts on the drawing. An alternative method is to draw just a portion of the input text on the drawing. This technique relies on the fact that comments can be nested in HCL. To draw just some of the modules contained in the input text, the engineer includes the modules that won't be shown in comments. HCL will ignore the module definitions contained in comments.

A third method is to "pull apart" modules. One problem that can occur with large charts is that some of the subcharts may be large compared to other modules. The larger subcharts can be divided into multiple subcharts to provide a better looking drawing. This is done by making a module called within the larger subchart a utility module. A module can be made a utility module simply by listing it more than once in the input text.

Additional Output from HCL

There are several features of HCL that general-purpose graphics editors do not provide. These outputs are intended to help the engineer during the design of the software.

The module call count gives an alphabetic listing of all the declared modules, their types, and the number of times each is called. If a module is declared but never called, it is flagged to bring it to the attention of the engineer, who may have forgotten to use the module after declaring it. At the end of the listing is the total count for each type of module as well as the total number of modules declared. This output can be used to find the critical modules that are called often in the designed software. This output may also be used in the collection of certain software metrics.

The following is an example of a module call count table.

Module Call Count Table

A	(M)	called	1 time.
B	(M)	called	1 time.
C	(M)	called	1 time.
D	(M)	called	1 time.
E	(M)	called	1 time.
F	(S)	called	3 times.
Z	(R)	******Not called******	

Module (M)	5
System (S)	1
Recursive (R)	1

Total Modules Declared = 7

HCL produces two types of cross reference outputs. The first is an alphabetic listing of all the modules and the modules they call (see example below). Included with this are the parameters that are used in each call, which is especially helpful if the parameters on the chart become too small to read.

Module Call Cross Reference

Module	Calls Modules
A	B (VAR1, VAR2) C D(/VAR3) E
B	F
C	
D	F
E	F
F	
Z	

Number of modules declared = 7

The second cross reference, shown below, is the reverse of the first one. It is also alphabetical, but shows all the modules that call a given module. This comes in handy when the designer wishes to change the interface of a module and needs to know which other modules will be affected.

Module Called by Cross Reference

Module	Called by Modules
A	
B	A
C	A
D	A
E	A
F	B, D, E
Z	

Number of modules declared = 7

Uses of HCL

Although HCL was designed as a software engineering tool, it is not restricted to this use. Other uses include:

- File system map. A hierarchical file system such as MS™-DOS or the HP-UX operating system can be represented.
- Management organization chart. A company organization chart can easily be created and maintained.
- Process mapping. A process can be decomposed and documented easily with HCL.

Other uses include specifying data structure composition, documenting a command tree, and depicting Modula-2 interdependencies. Basically, any hierarchically organized structure can be documented using HCL.

Conclusion

HCL has been used successfully on a variety of software products within HP. These projects have ranged from real-time microprocessor-based assembly language to the HCL program itself. Two such projects were described in the March 1985 issue of the *HP Journal*—the HP 7978A and HP 9144A Tape Drives.[4] HCL was indispensable in the execution and management of these projects.

HCL has proved to be very easy to learn. An engineer who is familar with the text editor can be productively generating hierarchy charts in less than a half a day. HCL has vastly decreased the nonproductive tasks associated with hierarchy charts, freeing the engineer to concentrate on software design.

While HCL can be ordered inside of HP, it is an internal tool and is not, at this time, available for customer sales.

Acknowledgments

We'd like to acknowledge Frances Cowan and Perry Wells, who provided many hours of testing and supplied useful enhancements that were incorporated into HCL. We would also like to extend our thanks to Ellen Brigham and Phiroze Petigura for arranging internal distribution of the product.

References

1. W.P. Stevens, *Using Structured Design*, John Wiley and Sons, 1981.
2. G.J. Meyers, *Composite/Structured Design*, Van Nostrand Reinhold Company, Inc., 1978.
3. H. Yourdon and L.L. Constantine, *Structured Design*, Prentice-Hall, 1979.
4. *Hewlett-Packard Journal*, Vol. 36, no. 3, March 1985.

Module Adds Data Logging Capabilities to the HP-71B Computer

This 64K-byte plug-in ROM offers new BASIC language keywords for control of a battery-powered data acquisition and control unit and nine application programs for data capture, presentation, and transmission to host computers.

by James A. Donnelly

THE COMBINATION OF THE HANDHELD HP-71B Computer[1] and the HP 3421A Data Acquisition/Control Unit[2] provides a low-cost hardware configuration for many engineering or production data acquisition applications (Fig. 1). The computer and instrument are connected via the Hewlett-Packard Interface Loop (HP-IL).[3] To assist the engineer in performing data acquisition tasks, a special plug-in ROM module was developed for the HP-71B Computer. This 64K-byte ROM module, the HP 82479A Data Acquisition Pac, contains a hybrid of BASIC and assembly language programs. Six general sets of capabilities are provided:

- BASIC keywords for instrument control. The keyword INIT3421 finds and initializes the specified HP 3421A on the interface loop. Keywords such as DCVOLTS and RANGE provide convenient instrument control. Additional keywords such as TCOUPLE and RTD provide rapid and accurate assembly language linearizations for thermocouple, thermistor, and resistance-temperature detector (RTD) probes.
- Interactive control of the HP 3421A. A BASIC program and keyboard overlay for the HP-71B create a virtual front panel for the HP 3421A, which has no front-panel controls.
- Nine-trace stripchart output for the HP ThinkJet Printer.[4] A BASIC program configures the system to produce strip charts with optional data storage.
- System monitoring and control. A BASIC program configures the system to monitor functions in a system, perform limit tests and controls, and display the system status on a video interface. An option allows periodic storage of the system status to a data file.
- Long-term data acquisition and control. Two BASIC programs allow sophisticated data logging and control pro-

Fig. 1. *The HP 82479A Data Acquisition Pac for the HP-71B Computer enables the computer to control the HP 3421A Data Acquisition/Control Unit via the Hewlett-Packard Interface Loop, allowing an engineer to configure low-cost, battery-powered systems for data logging or instrument control.*

cedures to be configured and executed without user intervention.

- Data analysis. A BASIC program provides printed analysis of data collected by the stripchart, system monitor, or logging programs. The data can be printed, summary statistics can be calculated, or a strip chart can be generated from stored data. Two additional programs provide data transmission to MS™-DOS-based computers (via HPLink) or to HP 9000 Series 200 and Series 300 Computers.

New BASIC Keywords

The Data Acquisition Pac's capabilities are based on a series of BASIC language keywords that combine conventional instrument control steps into one action. The conventional procedure for reading an instrument in HP BASIC languages has been to use the OUTPUT statement to send a command sequence to the instrument and then use an ENTER statement to receive the data from the instrument. The keywords provided in the HP 82479A ROM combine these operations into one, which provides several benefits:

- Ease of programming: the engineer is no longer required to refer to the instrument manual for cryptic commands. For example, A=DCVOLTS replaces OUTPUT :6;"F1T2"@ ENTER :6;A
- Enhanced code maintainability: an engineer assigned to take over responsibility for a test program using these keywords will experience a shortened learning curve while reviewing the code.
- Speed enhancement: the combined operations reduce operating system overhead for the processing of the OUTPUT and ENTER statements. The keywords TCOUPLE, THMST2, and RTD provide rapid and more accurate conversions from voltage or resistance measurements than equivalent routines written in BASIC.
- Device location independence: unlike the HP-IB (IEEE 488), where the addresses of the instruments must be set manually, the HP-IL assigns device addresses automatically. The keywords in the Data Acquisition Pac complement this by not requiring instrument address information. If more than one HP 3421A is connected to the HP-IL, a consistent device addressing scheme makes selection of the first instrument on the loop the default choice, but permits selection of additional HP 3421As.

The new keywords provided by the ROM do not preclude the use of ENTER and OUTPUT statements to control the HP 3421A, creating a possible conflict between commands issued through the new keywords and commands sent with the OUTPUT statement. This potential for conflict is virtually eliminated by assigning priority to the commands made with the new keywords and keeping track of the intended state of the HP 3421A in an internal buffer in the HP-71B. The buffer records the current settings for:

- The HP 3421A device specifier (address)
- The degree mode for temperature conversions (C, F, K, or R)
- The gate time for the counter (0.1, 1, or 10 seconds)
- The number of digits of resolution (3, 4, or 5)
- The range (−1 through 7)
- The autozero status (on or off)
- The autorange status (on or off).

The entire instrument buffer is sent to the HP 3421A before a new reading is taken, ensuring that the instrument will be in the correct state. This way the settings requested by the new keywords in the Data Acquisition Pac will be enforced even if all the instrument settings have been altered by an OUTPUT statement in another program.

The HP 3421A can be ordered with various optional plug-in cards, depending on the needs of an application. This means that some commands will be correct for specific configurations, but incorrect for others. To facilitate rapid error detection, the instrument status is checked after each command is sent, so that problems such as invalid channel requests or ranges can be detected immediately. This saves another check that would need to be done from a BASIC program using ENTER and OUTPUT statements to talk to the HP 3421A.

The keywords provided for conversions from voltage or resistance to temperature are based on curve fits originally written in BASIC for a desktop computer. The BASIC routines provided accurate results, but with a significant speed penalty. These routines were rewritten in assembly language, providing a 20-to-1 speed improvement while minimizing the effect of round-off errors by using 15-digit internal math routines.

To complement the calculator-like friendliness of the HP-71B operating system, temperature unit conversion routines are built into the keywords. The HP-71B has a variety of system settings, such as OPTION BASE for array declarations and OPTION ANGLE for trigonometric functions. The OPTION statement was extended to include OPTION DEGREES unit. The available temperature units are Celsius, Fahrenheit, Kelvin, and Rankine. By declaring OPTION DEGREES C, the programmer specifies that the results of future temperature conversions will return Celsius degrees.

Binary Subprogram

A binary subprogram called SCAN was written to complement the 30-reading data buffer in the HP 3421A. This subprogram provides significant performance enhancements by replacing the entire command sequence and loop structure normally used in BASIC for a burst measurement into a single binary subprogram call. The subprogram requires a command string that specifies the measurement, a vector to retain the collected readings, an index that points into the vector to indicate the starting position for the readings, and an error parameter.

BASIC Programs

The BASIC programs are designed to take full advantage of the features in the HP-71B operating system, yet retain the friendly personality of a calculator. To this end, the front panel and data logging programs are designed to work with or without peripheral devices such as printers or video interfaces. The user interfaces of the various programs are designed to be consistent, so that an operator familiar with one program will feel at home with another. Many complex operations requiring a number of commands on larger computers are reduced to a single keystroke. Error handling is designed to reduce the impact of simple entry mistakes or requests for impossible measurement or control functions.

System Monitor Example

The combination of the HP-71B Handheld Computer, the HP 3421A Data Acquisition/Control Unit, and the HP 82479A Data Acquisition Pac can be used to provide monitoring and control functions in space-critical or budget-critical environments where a large system simply may not fit. The MONITOR program can not only monitor a system, but can also control the system using limit tests. Consider a production photographic laboratory with a controlled-temperature bath that must be maintained within one degree of 25°C. Two type-T thermocouples are used to monitor the temperatures. A room-temperature thermocouple is connected to channel 3 of the HP 3421A, and a bath-temperature thermocouple is connected to channel 4. The bath heater is controlled by a relay, which in turn is controlled by actuator channel 0 of the HP 3421A. The HP-71B containing the HP 82479A plug-in ROM is connected to the HP 3421A and to an 80-column video interface and video monitor as shown in Fig. 1.

In this example, the MONITOR program continuously displays the room temperature and bath temperature on the video monitor. In addition, two sets of limit tests are specified: the control limits and the alarm limits. The lower and upper control limits are set at 24.5 and 25.5°C. When the temperature falls below 24.5°C, the actuator channel is closed, turning on the heater by means of a relay. When the temperature rises above 25.5°C, the actuator channel is opened, turning the heater off. The lower and upper alarm limits are set at 24 and 26°C. If the control system fails, one of the alarm limits will be reached and the HP-71B will beep. Additional alarm limit actions could log the event on a printer or control additional actuators.

Fig. 2 illustrates the contents of the video monitor display while the photographic laboratory monitor is running.

PHOTOLAB TEMPERATURE MONITOR

Setup:	Datefile:	TESTDATA:MAIN	Time: 14:22
1 Room Temp.	21.85	I X	I
2 Bath Temp..	25.17	I X	I

Fig. 2. *Video display for system of Fig. 1.*

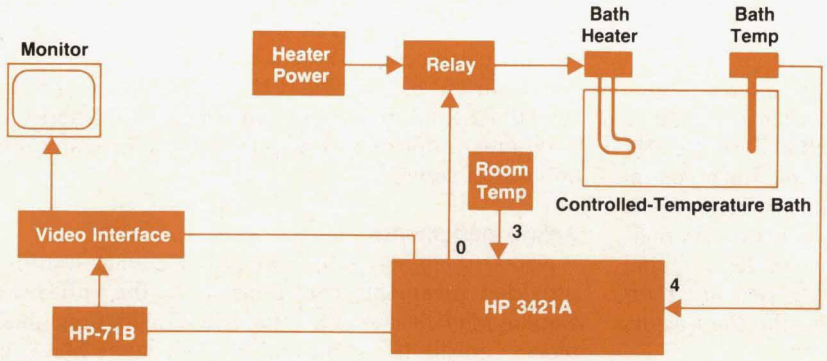


Fig. 1. *Photographic laboratory temperature monitoring system with actuator control.*

Wherever possible, the user's working environment is preserved to protect the value of working variables, files, and other data.

Unlike larger desktop computers, the HP-71B is designed to work under extremely low memory conditions while managing multiple data and program files in memory. Hence, the data logging program options are designed to work under low memory conditions. In the event of equipment failure, data is always preserved. The file update procedures to external mass storage devices are designed to protect the integrity of the file at the slight cost of processing speed. An HP-IL failure during disc access risks at most one data scan, leaving the other records intact. The data analysis and transfer programs are designed to accommodate data files with partial data, such as data from an experiment that terminated with an equipment failure or upon receiving an abort command from the operator.

The BASIC programs in the HP 82479A Data Acquisition Pac are described below:

- The FRONT program in conjunction with a keyboard overlay redefines the HP-71B keyboard, mapping the HP 3421A functions to individual keys. Additional key redefinitions combine voltage or resistance measurements with temperature linearization functions to provide complete temperature measurement functions.
- The STRIP program in conjunction with an HP ThinkJet Printer produces up to nine traces on a 1% scale. The measurement function and scale are specified for each trace. The scales for each trace are printed at the top of each page of output. Program options include measurement interval times and data storage options.
- The MONITOR program in conjunction with an HP-IL video interface provides a visual system monitor and control capability. Up to 18 traces can be presented on the video display. A trace consists of a horizontal line on the display showing a label, an actual measurement, and a 32-point scale indicating the position of the current measurement relative to specified nominal bounds. The display is designed so that the system operator can assess the state of the system at a glance, instead of having to interpret numerical readings one by one. Five limit tests can be applied to the measurement of each trace. If a test fails, one of six available limit actions can be taken.
- The SETUP program is used to define a data logging procedure. Up to 20 groups can be specified for a single data logging experiment. A group definition consists of the mea-

surement function specification, up to five limit tests, and data storage and timing specifications. The LOG program is used to execute the data logging setup. Options in the LOG program include buffered data storage and device power-down capability for extending the life of battery-powered peripherals.

- The REPORT program provides printouts of collected data, summary statistics about the data, and strip charts from collected data. These options can examine the entire file or a time segment within the file.
- The TRANSFER and MSDOSXFR programs are used to move collected data to Series 200 and Series 300 Computers or MS-DOS-based computers such as the HP 150, The Portable, or the Vectra. Files transferred to the MS-DOS computers are compatible with 1-2-3™ from Lotus™.
- The STATUS program is used to read the status registers in the HP 3421A and produce a comprehensive report listing error conditions, option configurations, and the current operating status.
- The VERIFY program provides an interactive diagnostic procedure for verifying the proper operation of the HP 3421A. The program prompts for the installation of a diagnostic block on each option board installed, and checks for proper operation with the diagnostic circuits.

Measurement Options

Twenty-one measurement functions are offered among the three main data acquisition programs. These functions correspond to the main capabilities offered by the HP 3421A combined with the temperature linearization keywords. The functions include dc volts, ac volts, direct current, two- and four-wire resistance, frequency, six thermocouple types, 2-kΩ and 5-kΩ thermistors, RTD, digital bit, and digital byte. The programs provide two- and four-wire resistance measurement options for the thermistors and the RTD.

Clearly, the BASIC programs cannot anticipate all possible measurement applications involving the HP 3421A. An additional function is included that permits the user to write a special BASIC subprogram to perform custom measurement procedures. This hook allows new measurement procedures to be created that still take advantage of the user interface and data storage facilities provided by the programs in the Data Acquisition Pac.

In addition, each function can call a conversion program for additional processing of a measurement. For instance, there is no alternating current function in the HP 82479A ROM. A simple conversion program that divides an ac voltage by the shunt resistance can provide the equivalent of an alternating current function.

Limit Tests

The MONITOR and LOG programs can perform limit tests on data collected by each function. A simple negative feedback loop can be created for temperature control by setting a limit test that turns on a heater if a temperature falls below a set level, or turns off the heater if the temperature rises above a certain level. Limit actions include a simple beep, the printing of a message, switching an actuator, enabling or disabling another measurement group (in the case of the LOG program), or the calling of a user-written program. As mentioned before, the programs cannot anticipate all of the possible actions that might have to take place in the event of an out-of-limit condition. A hook that allows the user to write a custom limit action program provides significant flexibility in system design.

Error Recovery

Most of the peripherals that are available on the HP-IL are battery-powered and not subject to the misfortunes of ac power line interruptions. Nevertheless, under some conditions a device may temporarily malfunction or cease to operate, causing an error to be detected by the HP-71B. In most instances, the execution of a RESTOREIO command is sufficient to return the interface loop to working order. Clearly, in either production or unattended long-term data acquisition applications, some form of automatic error recovery is desirable. Each of the programs in the Data Acquisition Pac calls a subprogram RECOVER when a loop problem is encountered. The recovery subprogram is sufficient to bring the system back to working order in many cases without operator intervention. Some system configurations may require different error recovery procedures than are provided. By placing a new RECOVER subprogram in the memory of the HP-71B, the user can effectively replace the one in the HP 82479A module. This hook provides more sophisticated system designs for error recovery. For example, if an HP 82402A Dual HP-IL Adapter is installed in the HP-71B, the recovery subprogram might notify a host computer connected to loop two that loop one is broken and out of service.

Acknowledgments

Nathan Zelle wrote the assembly language routines and provided invaluable contributions to the entire project. Nathan Meyers assembled the data transfer programs from pieces contributed by me and Bill Saltzstein. Many people expressed interest and offered design suggestions, making this a true "next bench" project. Notable for their contributions are Bob Botos, Jerry Hamann, and Peter Vanderheiden of the Loveland Instrument Division and Grant Garner, Dan Rudolph, and Don Ouchida.

The review and testing of this project were large efforts. Many thanks are due to Mark Banwarth, Dirk Bodily, Dave Boggan, Chris Bunsen, Jennifer Burnett, Ron Henderson, Tim Hubley, Michel Maupoux, Pat Megowan, Henry Nielsen, Dan Parker, and Don Rodgers.

References

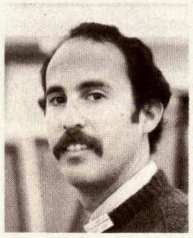
1. Complete issue, *Hewlett-Packard Journal*, Vol. 35, no. 7, July 1984.
2. J.J. Ressmeyer, "Low-Cost and Portability Come to Data Acquisition/Control Products," *Hewlett-Packard Journal*, Vol. 34, no. 2, February 1983.
3. R.D. Quick and S.L. Harper, "HP-IL: A Low-Cost Digital Interface for Portable Applications," *Hewlett-Packard Journal*, Vol. 34, no. 1, January 1983.
4. C.V. Katen and T.R. Braun, "An Inexpensive, Portable Ink-Jet Printer Family," *Hewlett-Packard Journal*, Vol. 36, no. 5, May 1985.

Authors

March 1986

4 AI Workstation Technology

Martin R. Cagan



Interested in programming environments, software development methodologies, and computer-assisted instruction, Marty Cagan is a project leader in the Software Technology Lab of HP Laboratories. Joining HP in 1981, he has worked on business applications for the HP 3000 Computer and the implementation of the HP Development Environment for Common Lisp product. He holds BS degrees in computer science and economics awarded in 1981 by the University of California at Santa Cruz. A member of the ACM, the AAAI, and the IEEE Computer Society, Marty is a resident of Los Altos, California.

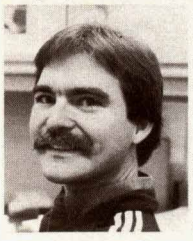
15 Defect Tracking System

Steven R. Blair

A native of Seattle, Washington, Steve Blair attended the nearby University of Washington, earning a BS degree in computer science in 1983. He then joined HP and is now part of the staff of Corporate Engineering. A member of the ACM, he lives in Santa Clara, California, and is interested in photography, hiking, and competitive sailing.

19 Object-Oriented Toolset

Gregory D. Burroughs



A native of San Francisco, Greg Burroughs studied mathematics at the University of California at Riverside (BS 1978 and MA 1979) and computer science at the University of Wisconsin (MS 1981). With HP since 1981, he has worked on computational geometry and software for digital circuit design and verification. He is currently involved with microprocessor design validation and testing. Greg is the author of three conference papers, one on function recognition in VLSI circuits and two on software engineering, and is a member of the Mathematics Association of America and the Society for Industrial and Applied Mathematics. Outside of work, he enjoys singing in his church's choir, performing Renaissance music, and playing volleyball. Greg lives in Sunnyvale, California.

24 Software Test Automation

Craig D. Fuget

Craig Fuget was born in Pittsburgh, Pennsylvania and studied computer science and engineering at the Massachusetts Institute of Technology. He completed work for his BS degree in 1983. He joined HP the same year and is a software quality engineer responsible for metrics, testing, and tools, primarily for the operating system for the HP 1000 Computer. He is a member of the IEEE. Craig lives in Palo Alto, California and likes reading, traveling, camping, and other outdoor activities.

Barbara J. Scott

Barbara Scott studied computer science at the University of California at Davis, earning her BS degree in 1979 and her MS degree in 1980. With HP since 1980, she is a project manager for HP-UX system testing. She has also tested operating systems and has been the technical leader responsible for the development of tools, processes, and training designed to improve HP programmer productivity and software quality. Barbara was born in Denver, Colorado and now lives in Sunnyvale, California with her husband. She likes racquetball, aerobics, sewing, and camping.

28 Software Quality Metrics

William T. Ward

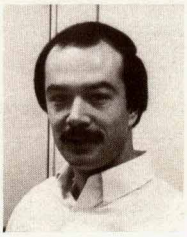
With HP's Waltham Division since 1982, Jack Ward is a software quality assurance engineer. He was responsible for testing the software for the HP 78720A ECG Arrhythmia Monitor. He has also been a technical marketing engineer and was a software support engineer for Data General Corporation. His academic background includes a BS degree in linguistics from the University of Illinois in 1972 and an MS degree in computer science from Boston University in 1984. A resident of Brookline, Massachusetts, he also teaches graduate-level computer science courses at Boston University. Jack is married, has one child, and has another on the way. He enjoys jogging and keeps himself busy by renovating a recently purchased 100-year-old Victorian home.

32 P-PODS

Robert W. Dea

With HP since 1979, Bob Dea was born in Pittsburg, California and attended the University of California at Berkeley. He received his BS degree in electrical engineering and computer science in 1973. He has contributed to the development of software for the HP 3000 Computer and design tools such as P-PODS. Before coming to HP, he worked in the aerospace industry. He is the coauthor of an earlier HP Journal article. Bob lives in Fremont, California, is married, and is interested in meditation and martial arts. He also enjoys fishing, photography, and personal computers.

Vincent J. D'Angelo



Vince D'Angelo studied computer science at California State University at Chico and received the BS degree in late 1978. He then came to HP and has worked on a number of applications and tools for the HP 3000 and HP 9000 computers. He was the project leader for P-PODS and is currently working on software design support tools. Vince previously worked for Burroughs Corporation on application development. He lives in Sunnyvale, California, plays the violin, and is involved in numerous church activities.

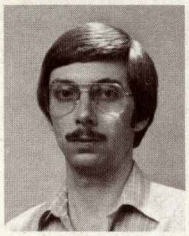
35 Triggers

John R. Bugarin

With HP since 1981, John Bugarin is a software productivity manager at HP's Colorado Networks Operation. He has also been a project manager responsible for the development of local area networks for the HP 9000 computer family. John was born in Milwaukee, Wisconsin and attended the University of Wisconsin at Madison. He completed work for his BS degree in computer science and mathematics in 1979 and for his MS degree in computer science in 1980. He's a member of the ACM. He lives in Fort Collins, Colorado and likes skiing and golf.

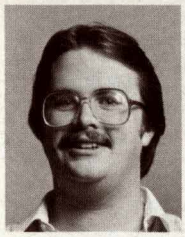
37 Hierarchy Chart Language

Bruce A. Thompson



Born in Manchester, Iowa, Bruce Thompson studied computer engineering at Iowa State University and received a BS degree in 1981. He then joined HP's Greeley Division where he has worked on the software design for the HP 7974 and HP 7978 Tape Drives and the HP 88500 Disc/Tape Interface Card. He is the coauthor of two conference papers on hierarchy charts and parallel processing using microcomputers. He also wrote one paper on structured analysis. A resident of Fort Collins, Colorado, Bruce is an amateur radio operator (WB0QWS) and likes skiing, photography, and watching football.

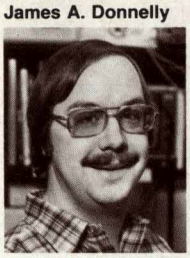
David J. Ellis



A coauthor of two conference papers on hierarchy charts and parallel processing using microcomputers, David Ellis has worked on the software design for the HP 7978 Tape Drive and the HP 88500 Disc/Tape Interface Card since joining HP in 1983. Before that, he wrote compilers, assemblers, and linkers for Tektronix, Inc. after completing the requirements for a BS degree in computer engineering at Iowa State University in 1981. David was born on Scott Air Force Base in Illinois and now lives with his wife and two daughters in Fort Collins, Colorado. He enjoys listening, playing, and singing numerous types of music and is involved in learning how to ski without injuring himself.

43 HP-71B ROM

James A. Donnelly



With HP since 1981, Jim Donnelly was born in Chicago, Illinois and attended Oregon State University. He completed work for a BS degree in broadcasting in 1979 and did instrumentation programming at the University of Oregon as well as running his own software consulting business before joining HP. He is now an R&D engineer and contributed to the design of the ROM for the HP 71B Handheld Computer. He is also the coauthor of three technical papers. Jim lives in Corvallis, Oregon and is a member of the Corvallis Art Guild. He likes to travel in the American West and northern Europe and enjoys music, photography, and cars.

HEWLETT-PACKARD JOURNAL

March 1986 Volume 37 • Number 3

Technical Information from the Laboratories of Hewlett-Packard Company
Hewlett-Packard Company, 3000 Hanover Street
Palo Alto, California 94304 U.S.A.
Hewlett-Packard Central Mailing Department
P.O. Box 529, Startbaan 16
1180 AM Amstelveen, The Netherlands
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

HEWLETT PACKARD

5953-8545