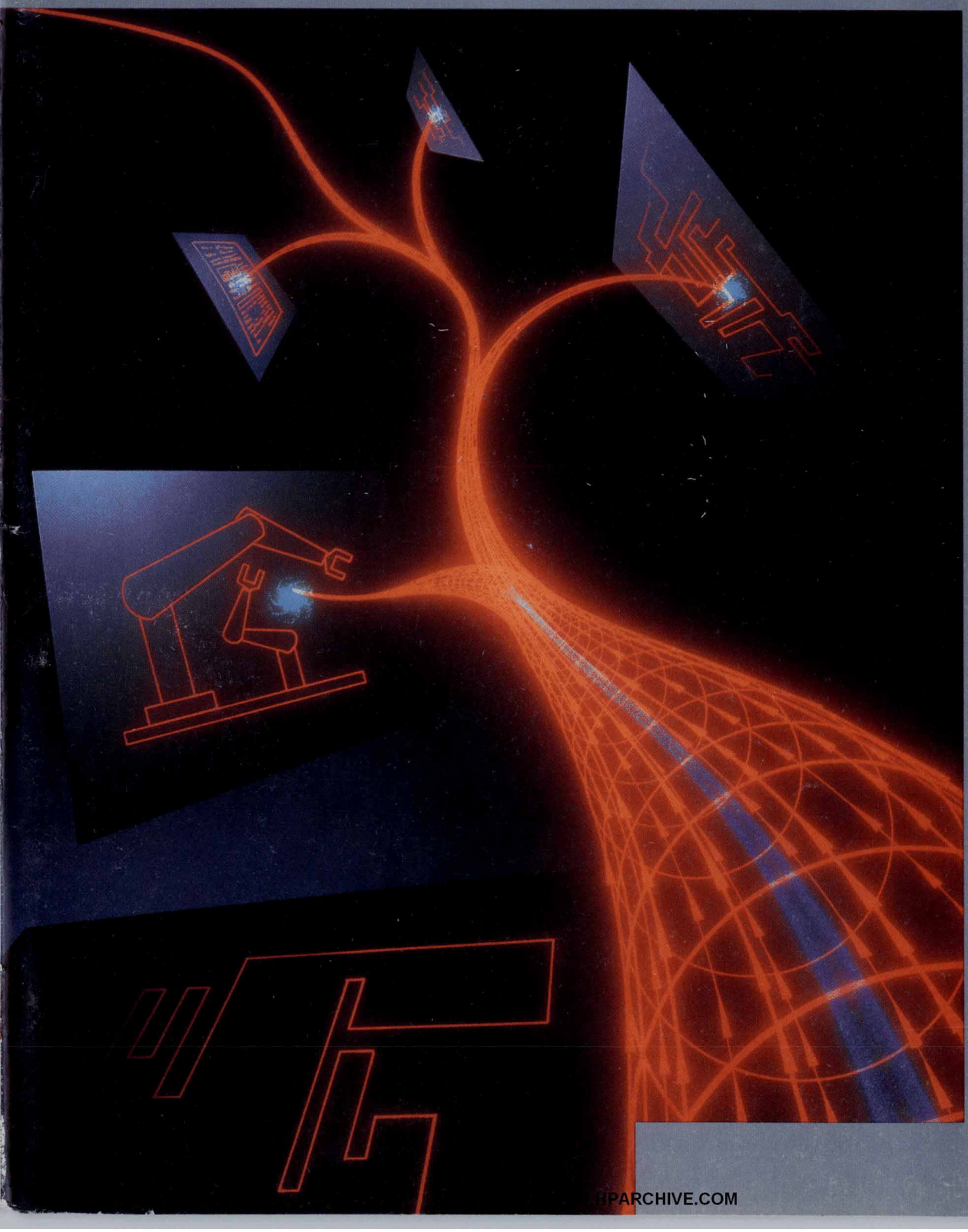
HEWLETT-PACKARD
JOURNAL

December 1991



HEWLETT PACKARD

HEWLETT-PACKARD
JOURNAL

December 1991

Articles

6 **HP Software Integration Sockets: A Tool for Linking Islands of Automation,** *by Mitchell J. Amino, Cynthia Givens, Mark Ikemoto, Alan C. Miranda, Scott A. Gulland, Kathleen A. Fulton, and Irene S. Smith*

13 **Configuration Files**

16 **Performance in the HP Sockets Domain**

20 **HP Sockets Gateway**

24 **Rigorous Software Engineering: A Method for Preventing Software Defects,** *by Stephen P. Bear and Tony W. Rush*

32 **Specifying an Electronic Mail System with HP-SL,** *by Patrick G. Goldsack and Tony W. Rush*

38 **Specification of State in HP–SL**

40 **Specifying Real-Time Behavior in HP-SL,** *by Paul D. Harry and Tony W. Rush*

43 **History Specifications**

46 **Using Formal Specification for Product Development,** *by B. Robert Ladeau and Curtis W. Freeman*

51 **Formal Specification and Structured Design in Software Development,** *by Judith L. Cyrus, J. Daren Bledsoe and Paul D. Harry*

Editor, Richard P. Dolan • Associate Editor, Charles L. Leath • Art Director/Photographer, Arvid A. Danielson
Production Supervisor, Susan E. Wright • Illustration, Richard M. Dominguez • Typography/Layout, Rita C. Smith

59 **Telecommunications Network Monitoring System,** *by Nicola De Bello, Giuseppe Mazzucato, Antonio Posenato, and Marco Silvestri*

Departments

4 **In this Issue**
5 **Cover**
5 **What's Ahead**
66 **Authors**
69 **1991 Index**

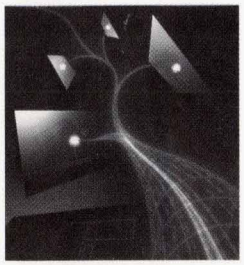
The Hewlett-Packard Journal is published bimonthly by the Hewlett-Packard Company to recognize technical contributions made by Hewlett-Packard (HP) personnel. While the information found in this publication is believed to be accurate, the Hewlett-Packard Company disclaims all warranties of merchantability and fitness for a particular purpose and all obligations and liabilities for damages, including but not limited to indirect, special, or consequential damages, attorney's and expert's fees, and court costs, arising out of or in connection with this publication.

Subscriptions: The Hewlett-Packard Journal is distributed free of charge to HP research, design and manufacturing engineering personnel, as well as to qualified non-HP individuals, libraries, and educational institutions. Please address subscription or change of address requests on printed letterhead (or include a business card) to the HP address on the back cover that is closest to you. When submitting a change of address, please include your zip or postal code and a copy of your old label. Free subscriptions may not be available in all countries.

Submissions: Although articles in the Hewlett-Packard Journal are primarily authored by HP employees, articles from non-HP authors dealing with HP-related research or solutions to technical problems made possible by using HP equipment are also considered for publication. Please contact the Editor before submitting such articles. Also, the Hewlett-Packard Journal encourages technical discussions of the topics presented in recent articles and may publish letters expected to be of interest to readers. Letters should be brief, and are subject to editing by HP.

Please address inquiries, submissions, and requests to: Editor, Hewlett-Packard Journal, 3200 Hillview Avenue, Palo Alto, CA 94304 U.S.A.

In this Issue



In the early days of factory automation, it was a lot easier to automate individual processes or workcells than an entire factory. The result was islands of automation, each representing the best available solution to an individual need but incapable of communicating easily with other islands or with higher-level systems because of differences in computers, interfaces, languages, operating systems, and applications. Yet the goal has always been the integrated factory, or CIM—computer-integrated manufacturing. The development of advanced networks and communication protocols such as MAP (see the August 1990 issue) has brought the integrated factory closer to realization, but it can still be a formidable task for a system integrator to provide for transparent communications between diverse applications designed for different computing platforms. HP Sockets, our cover subject, is a product that helps system integrators overcome incompatibilities and provide file and message transfer between applications running on different network nodes. HP Sockets runs on and provides communication between HP computers using the HP-UX and MPE XL operating systems. It can also communicate with non-HP systems through a gateway, an HP-UX node that uses a client/server model to extend HP Sockets capabilities to machines not using the MPE XL or HP-UX operating systems. Using HP Sockets, a system integrator creates application adapters, which allow applications to "plug into" the HP Sockets system. Data coming from an application is translated to an internal common data representation format. Outgoing data is converted from this format to the format expected by the destination application. The system provides a data manipulator to resolve differences in data structures and languages and a data transporter to move the data from origin to destination, guided by user-created configuration files describing the environment. The full story of HP Sockets design, operation, and performance can be found in the article on page 6.

With the use of structured analysis and structured design becoming commonplace in software engineering, where do we find the leading edge today in industrial software development methods? One range of methods that's definitely leading-edge is rigorous software engineering, also called formal methods—rigorous, abstract mathematics applied to software specification in an attempt to head off defects right at the start. HP Laboratories in Bristol, England has been working on formal methods for several years, and one of these methods—the HP Specification Language, or HP-SL—has reached a stage of development where it's ready to be used in real-world software projects. The idea is to develop an abstract but precise description of the behavior of the software system. This description can be reviewed to ensure that the system works properly and that defects are not introduced because of deficiencies in specifying the required behavior. Although software behavior can be specified using a natural language, it's difficult to do so unambiguously. Formal specification languages like HP-SL use special syntax and special symbols based on discrete mathematics to avoid the ambiguities of natural language. The motivation for this approach and an overview of the symbols and syntax of HP-SL can be found in the article on page 24. Examples illustrating the use of HP-SL are presented in the articles on pages 32 and 40, which show how an electronic mail system and a real-time alarm monitor are specified. In the articles on pages 46 and 51, software engineers from two HP product divisions relate their experiences with HP-SL in actual product development projects.

Network monitoring can mean many things, depending on the type of network and the reasons for monitoring. Some typical objectives of network monitoring are to measure traffic and plan for future needs, to determine the quality of service, and to identify and locate faults. More and more, network monitoring is a distributed process, with measuring devices permanently installed throughout the network reporting to a central computer. The HP E3500A telecommunications network monitoring system described in the article on page 59 is a distributed system for monitoring telephone networks, mainly outside North America and Japan, that use the 2-megabit-per-second primary rate interface and the CCITT (International Telephone and Telegraph Consultative Committee) R2 or Number 7 signaling systems. Peripheral units connected to the network collect and preprocess data on various network parameters and transmit their measurements to a central computer, which elaborates the data and provides the user interface. The parameters measured, some of which are specified by the CCITT, are related to traffic intensity and type, the quality of service, and network efficiency and use.

December is our annual index issue. You'll find the 1991 index on page 69.

R.P. Dolan
Editor

Cover

An artist's rendition of a typical HP Sockets domain (a less artistic rendition is given in Fig. 1 on page 6). The cable-like strand going through the center of the spiral represents a LAN and the panels connected to the strand represent diverse applications that are communicating with each other via HP Software Integration Sockets.

What's Ahead

The February issue will have several articles on the design of the HP 54600 digitizing oscilloscopes. These are 100-MHz oscilloscopes that have the rapid display update rate characteristic of analog oscilloscopes along with the data storage advantages of digital oscilloscopes. The HP LanProbe monitors and HP ProbeView software for distributed monitoring of local area networks will be described in another article. A tutorial paper on neural networks will present the basic concepts of these computing architectures.

HP Software Integration Sockets: A Tool for Linking Islands of Automation

The task of integrating diverse applications over a network of HP and non-HP machines is made easier with this software tool.

by Mitchell J. Amino, Cynthia Givens, Mark Ikemoto, Alan C. Miranda, Scott A. Gulland, Kathleen A. Fulton, and Irene S. Smith

HP Software Integration Sockets (HP Sockets) is a software tool that enables efficient and reliable integration of new and existing software applications in a network of different computer systems and diverse applications. HP Sockets is designed to help system integrators overcome problems that are common in software integration and difficult to solve. HP Sockets provides a comprehensive set of communication features that are both broad and deep. It is intended to fulfill the needs of interapplication communications for file and message transfer. Messaging functionality is particularly extensive for both random and sequential, synchronous and asynchronous, destructive and nondestructive data transfers. Communicating processes can be written without regard to language, computer type, or network topology. The same methods of communication can be used for either local or remote communication in a homogeneous (same machines and operating systems) or heterogeneous (different machines and operating systems) environment. Fig. 1 shows some typical applications that could be integrated with HP Sockets.

HP Sockets runs on HP 9000 Series 300, 400, 700, and 800 computers running the HP-UX* operating system, and HP 3000 Series 900 computers running the MPE XL operating system. Sockets can also communicate with non-HP systems (see "HP Sockets Gateway" on page 20).

After a brief overview, this article will describe the operation and implementation of the major components of HP Sockets.

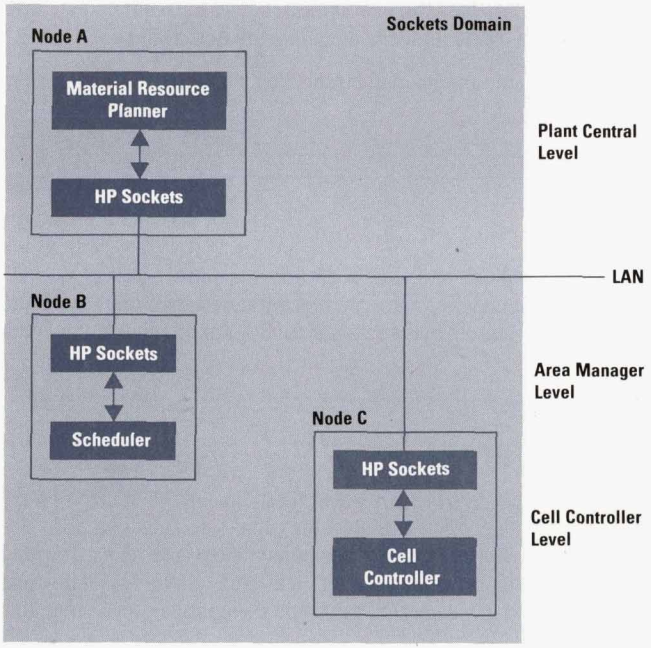


Fig. 1. Examples of the type of applications that might be integrated in the HP Sockets domain.

Overview

Using HP Sockets, a system integrator can shorten the system integration time by up to 75%. HP Sockets helps shield system integrators from all the vagaries of network interfaces and differences in hardware and system software. HP Sockets also helps resolve data incompatibilities between communicating applications. For end users, HP Sockets offers flexibility in the management of an integrated system. System configuration can be easily changed because HP Sockets supports incremental integration so that new hardware or software can be added without modification of the existing links.

The major problems involved in integrating new and existing software applications result from the differences in hardware platforms and their associated operating systems, and differences in the applications themselves. Specifically, these integration problems include:

- Differences in hardware platforms and operating systems. Because computers are built on diverse hardware architectures and operating systems, data from applications running on one system is often not usable on another system.
- Diverse network services. Programmers must change application code to create interfaces to different sets of network services.
- Incompatible data types. Different applications use different data types according to their specific needs. Programmers must write code to convert the data from a sending application into types that a receiving application can use.
- Differences in data structures. Incompatible data structures exist because applications group data elements in various ways. For example, an element with a common definition may be stored in two different ways: applica-

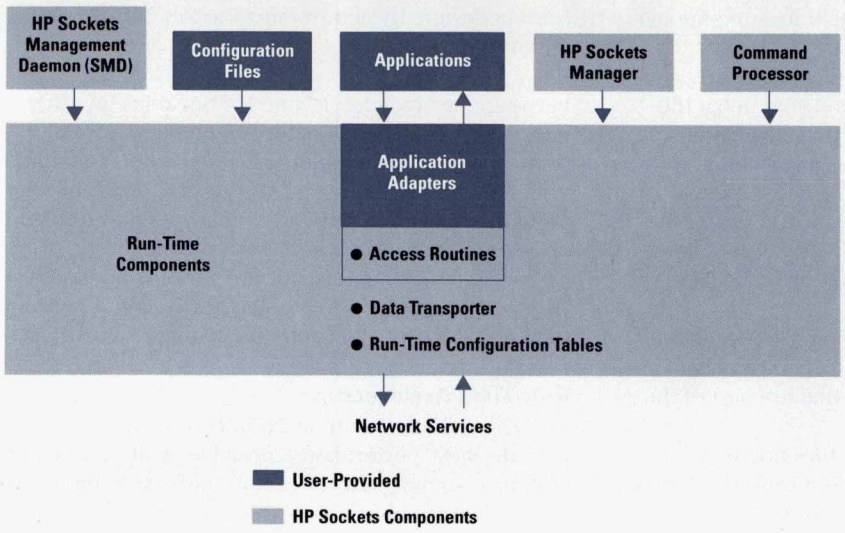


Fig. 2. HP Sockets run-time and administration components.

tion A may store it in one two-dimensional array and application B may store it in two one-dimensional arrays.

- Language differences. Applications written in different languages cannot communicate with one another if they represent the same data in different formats. For example, what is FALSE in FORTRAN can be TRUE in the C language.

HP Sockets helps to solve the above-mentioned problems by providing:

- Local and remote interapplication data transfer. HP Sockets supports links between existing applications with minimal or no modifications to the applications. It provides message and file transfers between applications, and data transfer can be synchronous or asynchronous.
- Configurable data manipulation. The user can resolve language, data format, and data type differences with data manipulation features such as rearranging, adding and deleting fields, and conversion between data types. These features are provided because of hardware, language alignment, and size differences between systems and applications.
- Local and remote program control. HP Sockets provides programmatic local and remote startup and shutdown of application programs.
- Simulated communication between applications. The HP Sockets command processor lets the user simulate communications between applications to test configurations and interfaces.
- Error logging. HP Sockets logs errors about data transfers for diagnostics and recovery.
- Centralized but movable HP Sockets administration. HP Sockets is administered from a single control point which can easily be moved at any time to prevent it from being a central failure point.

HP Sockets Components

HP Sockets provides the features described above with minimum system resource use while generating optimum performance. These features are provided by separating HP Sockets components into two categories: run-time and administration components (see Fig. 2). The run-time components handle the manipulation and transporting of messages according to information in the HP Sockets memory-resident configuration tables. These components come into play when HP Sockets is started and running. The administration components provide the developer with the tools to create, test, and administer the HP Sockets configuration.

Run-Time Components. The run-time components include:

- Data Manipulator. This component handles the manipulations that must be performed on data when the source data differs from the destination data in language or layout (see Fig. 3).
- Data Transporter. The data transporter provides reliable data transfer and communications with other nodes across the network.
- Run-Time Configuration tables. The run-time configuration tables are memory-resident tables containing configuration information such as nodes, processes, data manipulations, and links between applications. The information in these tables is derived from the data in the configuration files.
- Access Routine Library. The access routines are the programmatic interface to HP Sockets. To build an integrated system, a system integrator uses the access routines to create application adapters that connect users' applications to HP Sockets run-time components.
- Application Adapters. Adapters are procedures or programs that link the applications to the HP Sockets system. They are created using the access routines mentioned above.

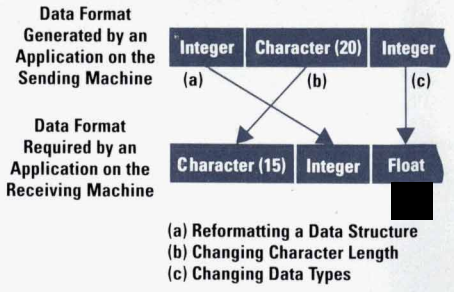


Fig. 3. Examples of data manipulations that might be required for applications to communicate when the machines, source languages, or applications differ.

Administration Components. The administration components include:

- HP Sockets Manager. The HP Sockets manager lets the user validate configurations, start up and shut down the HP Sockets system, use the command processor, and perform other HP Sockets management functions.
- Command Processor. The command processor allows interactive testing of the messages and links to any process within the HP Sockets domain. An HP Sockets domain is an environment consisting of all the nodes that are integrated via HP Sockets.
- HP Sockets Management Daemon (SMD). This process assists the HP Sockets manager in performing management tasks on local and remote nodes that are part of the HP Sockets domain.
- Configuration Files. The configuration files are user-created files containing information about the HP Sockets domain. At the startup of this domain, information in these configuration files is used to build the memory-resident configuration tables for the run-time part of HP Sockets (see "Configuration Files" on page 13).

System Integrators

As a software tool for system integration, HP Sockets is not a stand-alone solution. HP Sockets is one of many components that make up an integrated system. The participating applications play a major part. HP Sockets provides a foundation for the links between these applications, but these links have to be built by system integrators. The tasks the system integrator must perform to integrate applications with HP Sockets include:

- System analysis. The system integrator must determine the system functional requirements, analyze data flows between applications, determine data formats of the data flows, and define the characteristics of each link.
- System design. By determining the functional requirements and by analyzing the data flows and data formats, the system integrator will be able to design the bridges between the applications and HP Sockets—in other words, design the application adapters.
- Configuration. After building the application adapters, the next task for the system integrator is to configure the integrated system. Configuration means describing to HP Sockets the network topology, the participating processes, and the data formats and data manipulations that HP Sockets will need to perform for each link when it transfers data between communicating applications.

Run-Time Architecture

The HP Sockets run-time architecture is designed to provide the best performance possible while using minimal system resources. HP Sockets builds its run-time modules and activates them across the network without requiring user intervention.

Two main components make up the HP Sockets run-time architecture: the data transporter and the data manipulation module. Each component is implemented as one or more programs and consists of several modules (see Fig. 4). The architecture shown in Fig. 4 exists on every node integrated via HP Sockets.

Data Manipulation Module. On the sending node this module performs two operations: it manipulates outgoing data into a format acceptable to the receiving application and it encodes the data into a common data representation. The encoding operation is called *marshaling* and the common data representation is a language and machine independent data format. The incoming network manager on the receiving machine unmarshals the data (i.e., converts it to local machine and language representation).

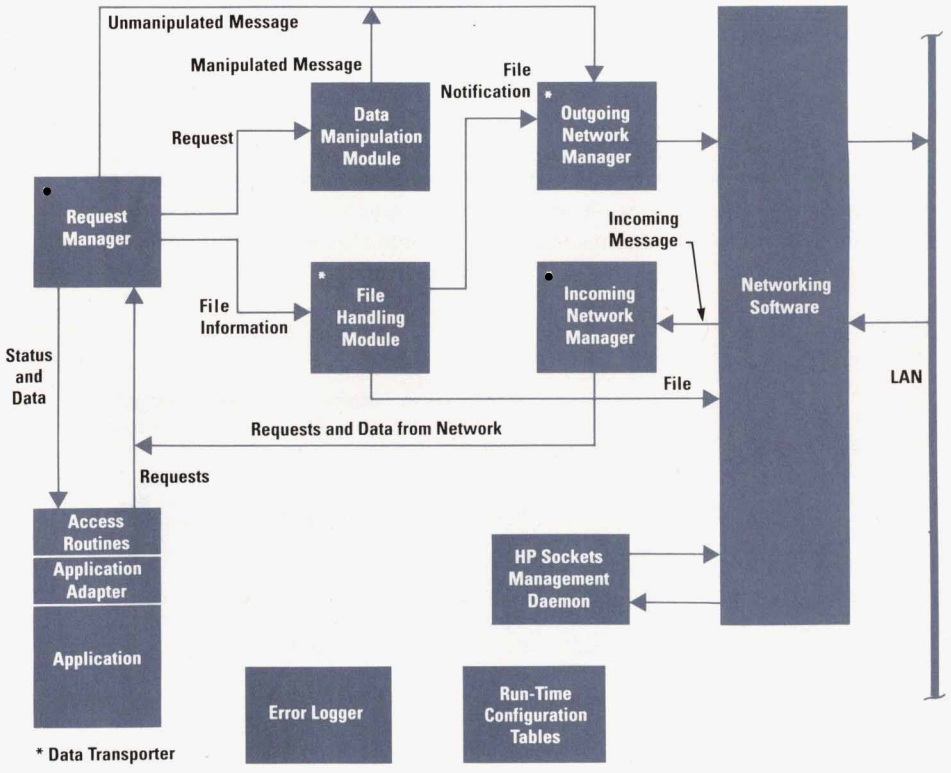


Fig. 4. HP Sockets run-time architecture.

For more about marshaling, common data representation, and data manipulation, see "Data Manipulation" on page 20.

Request Manager. This module routes application adapter requests through the internal HP Sockets modules to the request's eventual destination. The data messages it receives from the incoming network manager are stored until an application adapter issues a read request. The request manager handles only requests and leaves the data manipulation and file transfer tasks to other modules. Requests include message transfers, file transfers, and local or remote program control.

Outgoing Network Manager. This module sends messages over the network. It communicates with the incoming network managers on the other nodes. It creates and holds communication open to other nodes as required. Resource use is minimized because this module is shared by all the processes on a node, and it can clone itself as resources are needed.

Incoming Network Manager. This module receives messages from the outgoing network managers on other nodes. When it receives data it converts (unmarshals) the data from the common data representation that is used across the network into its host node's local representation. The data is passed to the local request manager for delivery. To optimize resource use, the incoming network manager also clones itself to meet its needs.

File Handling Module. This module transfers files between nodes. A process can request a file to be transferred and a notification to be sent to the request manager on the receiving node when the file arrives.

The error logger and the run-time configuration tables shown in Fig. 4 are used by both the data transporter modules and the data manipulation module. The error logger logs any errors or notable events (e.g., startups, shutdowns, etc.) detected during run-time operation. Access routines are provided so that the user can log user-defined messages using the error logger.

The run-time configuration tables contain data that HP Sockets extracts from the configuration files for run-time operation, such as node names and data manipulations.

Run-Time Operation

The run-time operation of HP Sockets on a sending node typically begins with the request manager receiving requests from an application adapter to perform a specific task such as to send a message or file, or to retrieve a message. Fig. 4 shows the data flows during these operations. If the request manager receives a request to send a message that is destined for a remote node and the message does not have to be manipulated, the message is sent directly to the outgoing network manager. When a message has to be manipulated, the request manager sends the message to the data manipulation module. After manipulating the message, the data manipulation module sends the message to the outgoing network manager. If the request manager receives a request to send a file, it notifies the file handling module, which sends the file and then sends a file notification message to the outgoing network manager. The outgoing network manager forwards the file message to the incoming network manager on the remote node.

On the remote node the incoming network manager sends the message to its request manager. If the file message has file notification turned on, the request manager sends a signal notifying the receiving application that a file has been sent to it. If file notification is not turned on, the application is not notified about the file sent to it. Status messages indicating success or failure of a request are returned to the sending application adapter when a request is made with wait.

Startup and shutdown are synchronized between the run-time modules. After the run-time modules are started, they perform initialization. All modules complete initialization and wait before they are allowed to process data so that if one module fails to initialize, startup can be aborted. Also, modules have some interdependencies that need to be resolved before data processing begins. For example, since modules rely on each others' system message facilities such as message queues, these facilities must be created by each module before data processing can begin. The HP Sockets management daemon is the process that notifies modules to start processing data. At shutdown, the HP Sockets management daemon notifies all modules to shut down. To prevent loss of messages in transit between modules, modules perform a graceful shutdown by sending last messages to the appropriate modules before terminating. The HP Sockets startup and shutdown functions are described in more detail later in this article.

Adapters and Access Routines

To link existing applications without requiring them to change, a bridge is needed between an application and HP Sockets. That bridge is called an application adapter (see Fig. 5). An adapter is usually a program, but it could be a procedure if the application is capable of calling user-written routines.

Application adapters are created with the access routines supplied by HP Sockets. This programmatic interface consists of HP Sockets routines callable from C, FORTRAN, COBOL, or Pascal programs to send messages, copy files, control processes, and so on. Any program or procedure that uses an HP Sockets access routine is, in effect, an

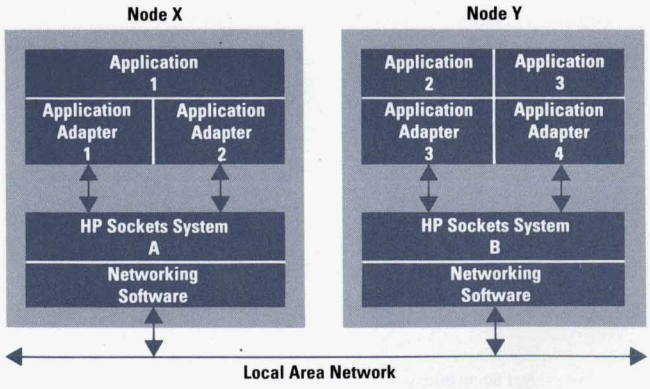


Fig. 5. Application integration with HP Sockets.

application adapter. The user determines how an application adapter is implemented.

There are two main functions in an adapter: data access and interfacing.

- Data Access. The data access function retrieves data from the application or delivers data to the application. Data access can be as simple as reading an application data file or accessing the application's data base.
- Interfacing. This means using HP Sockets access routines to transmit data between applications, or asking HP Sockets to start or stop another process.

Once the scope of the adapter is determined, the features of HP Sockets need to be factored into the design. The adapter will use these features through the HP Sockets access routines given in Table I. These routines are the door through which the HP Sockets world is entered. They are easy to learn, consisting of only 12 procedures which completely shield the interface developer from the network services levels. All functionality is expressed in functional terms such as read message, send message, send files, and so on. Sophisticated interfaces can be written with minimal programming.

HP Sockets provides a way to exercise these access routines interactively. This is done using the HP Sockets command processor. The command processor is a program that has a set of line mode commands that allow the user to simulate an adapter using access routines. Once HP Sockets has completed startup, with the command processor the user can read or send messages or files, and start or stop a user process. Data can be sent or displayed in either ASCII or binary.

The command processor can be used to test newly implemented adapters. Fig. 6 shows an example configuration consisting of two adapters, adapter 1 and adapter 2. Adapter 1 sends messages to adapter 2. If adapter 2 does not receive the message sent from adapter 1 the error can be found as follows:

- Test 1
  - Shut down adapter 1 and set up the command processor to simulate adapter 1.
  - Using the command processor, send a message to adapter 2 using the same format as would be sent by adapter 1. If adapter 2 receives the message correctly, then the defect is in adapter 1. If adapter 2 does not receive the message or receives an incorrect message the defect may be in the configuration or adapter 2.
  - Proceed to test 2.
- Test 2
  - Shut down adapter 2 and set up the command processor to simulate adapter 2.
  - Start up adapter 1 and receive the message with the command processor. If the command processor can receive the message correctly from adapter 1, the the defect is in adapter 2.

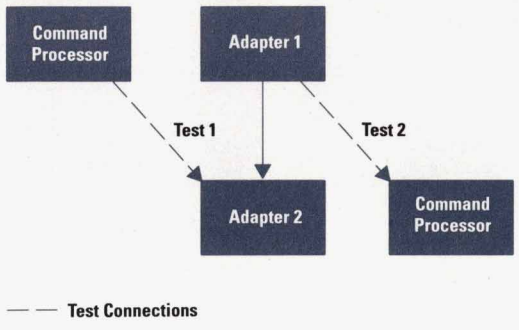


Fig. 6. Testing application adapters.

Table I
HP Sockets Access Routines

Access Routine	Description
These routines control adapters.	
SpInit	Initiate communication between HP Sockets and a calling process.
SpEnd	End communication and clean up resources allocated for the calling process by HP Sockets.
SpStartProcess	Start an HP Sockets configured user process.
SpStopProcess	Stop an HP Sockets configured user process.
SpErrLog	Log a user error message in the HP Sockets error logfile. (Send application-specific messages to the HP Sockets error logger.)
SpError	Retrieve ASCII messages corresponding to the HP Sockets error codes returned by routines.
These routines control data transfer between applications.	
SpSendMsg	Send a message to a local or remote process.
SpSendFile	Copy a file to a destination file on a local or remote node.
SpControl	Control the HP Sockets access routine options (such as timeout or message notification) that are available to the calling program.
SpReadQ	Read messages from the incoming message queue for the process.
SpFlush	Flush pending requests and clear the timeout list for the process. Use this only when doing a longjmp out of an interrupt routine.
SpDelGuaranteedMsg	Remove a guaranteed message from the guaranteed message spool file.

Data Transport

Data communication between an application and HP Sockets starts with the application adapter making a call to the the access routine SpInit to initiate communications with HP Sockets (see Fig. 7). This call sets up message transfer facilities (message queues for HP-UX and message files for MPE XL) for data and status messages between the application adapter and the request manager

and generates an initialize request to the request manager. The initialize request contains information such as the calling adapter's logical name, its process identification, and its newly created message queue identifier. The request manager enables the communication links for the calling process (whose name is already stored in the run-time configuration table) and returns a status message telling whether the SpInit call was successful. Once a process's communication link is enabled with a successful SpInit call, it can make calls to any of the other HP Sockets access routines.

The HP Sockets access routines handle all data transport between an application and the HP Sockets system. The main routines used for data transport are SpSendFile, SpSendMsg, and SpReadQ (see Table I). All access routines communicate between their calling program and the request manager, whose main function is to receive requests and route them between other internal modules, and between local and remote application adapters. As an example, the application adapter on the sending node initiates a call to SpSendMsg, sending:

- A data buffer
- A destination process logical name (on a remote node)
- An optional link logical name which indicates that data manipulation must be done before data goes to the outgoing network manager
- A no-flags-set parameter, which indicates send with wait.

After some preliminary error checking, the SpSendMsg parameters are put into the HP Sockets standard message structure and the message is sent to the request manager.

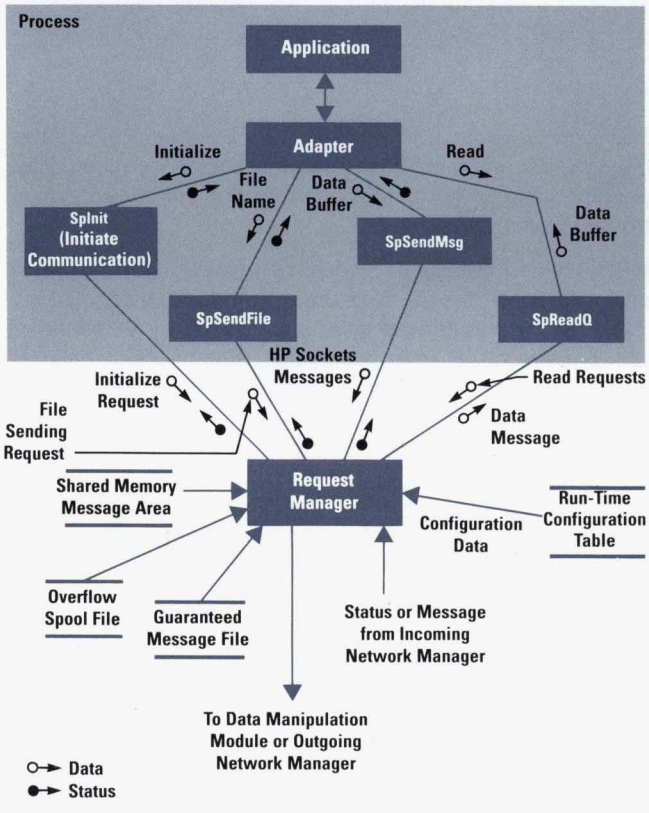


Fig. 7. Structure chart showing the communication data flows between the request manager and some of the HP Sockets access routines.

Since the routine was called with wait, the SpSendMsg call will not return to the calling routine until it either receives a status value back from the request manager or times out.

When the request manager receives the message from SpSendMsg, it validates the destination process logical name against the data in the run-time configuration table, and then checks to see if the destination process is configured to be on a remote node. If this is the case, the request manager sends a message to the outgoing network manager to set up communication with the node if it is not already done. Next, the logical link name is examined. If it is non-NULL, a message is sent to the data manipulation module, where the data is massaged according to the manipulations associated with that link name. After the manipulations are done, the message is encoded into the common data representation format and sent to the outgoing network manager for transfer to the remote node. If a link name had not been specified (i.e., a NULL link name), the message is sent directly from the request manager to the outgoing network manager, bypassing the data manipulation module completely.

When the request manager on the remote node receives a message (via the incoming network manager), it stores the message in its memory area. When the message is successfully stored, a return message containing the status of the transfer is sent back to the originating (local) node via the remote node's outgoing network manager. The local incoming network manager receives the return message, unmarshals it, and passes it to the local request manager. The local request manager sends it back to the adapter that originated the access routine call, completing the with-wait SpSendMsg call.

Once a data message arrives for a process on the destination node, the message is not sent to the destination adapter until the data is explicitly requested by a call to SpReadQ from the destination adapter. Until that call is made, data messages for all local processes are stored in an area of shared memory called the shared memory message area (an exception is guaranteed messages, which are described later).

The destination adapter has two ways of finding out if it has a data message waiting: polling and interrupt notification. Polling involves continually making calls to SpReadQ to see if there is a message. There can be a user-specified timeout for each of these calls. If a large enough timeout is specified, the adapter will block until a message arrives for it. Interrupt notification involves turning on the data or file notification feature of HP Sockets. When turned on, a notification (via a signal from HP-UX) is sent to the destination adapter when a data message or file arrives for that adapter. This feature is set up by the destination adapter through the use of the SpControl access routine. The adapter originating a SpSendFile or SpSendMsg call does not know whether the destination adapter has set up notification.

The SpReadQ message is sent to the request manager, which scans its memory area to find data messages associated with the calling adapter. If the correct data message is found (particular messages can be requested using

the SpReadQ tag parameters, or by specifying the logical name of the sending process whose message the receiving process wishes to read), it is removed from the shared memory message area and sent to the calling adapter.

If an incoming message needs to be put into the shared memory area, and either the whole memory area has been filled or the user-configurable percent allocation for a single process has been reached, the message is written to a special overflow spool file. This file will hold all subsequent messages for that process until space is freed in the shared memory message area by calls to SpReadQ. Anytime a successful SpReadQ is done for a process that has overflow messages, the overflow messages are read from the file into the shared memory area. This is done until the memory area limit is reached again or until the overflow spool file is emptied.

When the SpSendMsg access routine is called, a flag called the *guaranteed* flag can be set. Setting this flag tells the destination request manager to write the data message to a disk file rather than put it into the shared memory message area. The SpReadQ call done by the destination process must likewise set this flag so that the disk file is searched for the data message rather than the memory area. It is slower to send and receive a guaranteed message than a nonguaranteed message because of the overhead of disk access. However, the benefit of a guaranteed message is that the data will not be lost if a power failure occurs. The shared memory message area cannot be recovered after a power failure.

Guaranteed messages can be deleted using an option in SpReadQ, but to be absolutely sure that the data has been received the data should first be read with a flag indicating that the message is not to be removed from the disk file (using the no-consume flag), then as a separate step a call should be made to the SpDelGuaranteedMsg access routine to delete the message. A drawback of guaranteed messages is that they can only be read in the order they were received by the request manager; they cannot be retrieved by their tag values, or by specifying the logical name of the sending process.

Network Interface

The outgoing network manager and the incoming network manager are the modules within HP Sockets responsible for transferring messages across the network. Optimum performance is the reason network input and output tasks are divided between these two modules.

The incoming and outgoing network managers currently use NetIPC[1] intrinsics to interface with the network protocols. (BSD sockets can be used by HP-UX nodes running HP Sockets provided that there are no MPE XL nodes in the HP Sockets domain. MPE XL does not currently support BSD sockets.) With NetIPC, communication between the incoming and outgoing network managers involves establishing a virtual circuit connection, which is referred to as a virtual circuit socket descriptor. Multiple socket descriptors are needed for communication between multiple nodes. Socket descriptors are allocated

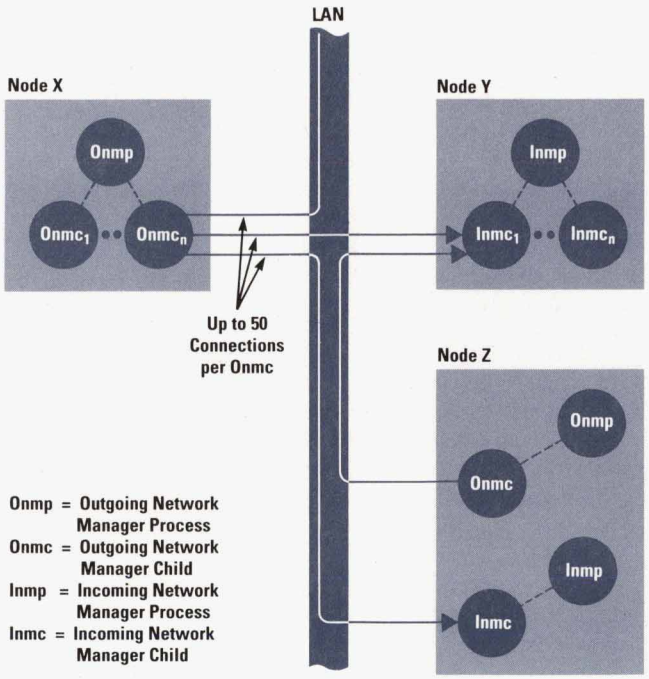


Fig. 8. Outgoing and incoming network manager processes and their connections.

from the same space as file descriptors. A process is limited to a certain number of file and socket descriptors that it can have open at any given time. The design of the outgoing network manager and incoming network manager allows HP Sockets to communicate with several nodes at a time by having the two network managers' parent modules (Onmp, Inmp) fork child processes (Onmc, Inmc) as needed to establish and hold connections (see Fig. 8).

When an Onmc or an Inmc reaches its maximum number of connections, another Onmc or Inmc process is created. The parents manage the children and do not hold any connections themselves.

When the request manager needs to send a message to a particular node for the first time, it notifies the Onmp, which in turn assigns an Onmc process the responsibility for communicating with that node. After the Onmc process successfully establishes a connection with the remote Inmc, they exchange a message to ensure that the HP Sockets configuration is a compatible version on both nodes. If the versions are not compatible, communication is terminated. Otherwise, the Onmc process can send messages it receives from local processes to the Inmc on the destination node. After receiving a message, the Inmc unmarshals the message and sends it to the local request manager, which is responsible for routing the message to the end application.

If an Onmc fails to establish or maintain communication with a node for any reason, such as a node or network failure, it spools to disk all messages that the sending application specifies as critical and that are destined for the node associated with the failure.

(continued on page 14)

Configuration Files

Configuring an HP Sockets system requires the user to create six configuration files. These six files, which can be created with any text editor, describe to HP Sockets in complete detail the HP Sockets operating environment with information such as the participating nodes, the communicating processes, and the data to be transported. Fig. 1 shows the six configuration files and the specific items of information that associate each file with the other files. These files can be created in any order using the information gathered in the system analysis phase. Only the first two files, NETWDEF and PROCDEF, are required to describe an HP Sockets domain.

Network Definition File (NETWDEF). The NETWDEF file identifies the nodes (computer systems) in the HP Sockets domain, giving their logical and physical names, hardware types, startup status, and C compiler availability (see Fig. 2). Nodes are identified by their physical LAN name and are given a logical name (e.g., **cpu1** in Fig. 2). All references to a node are done with the logical name so that if a node is removed, all HP Sockets functionality can be transferred to another name with just a change in the NETWDEF file. The startup status says whether or not a node should be started up.

Process Definition File (PROCDEF). The PROCDEF file lists all processes known to the HP Sockets domain (see Fig. 3). This file also ties a process to a logical node name (e.g., in Fig. 3 process **lookupcp** is tied to node cpu1). When the user specifies the movement of data in the system, the receiving process name is used, not the node name. The correct node is identified using the PROCDEF data. For processes listed in this file that are to be started or stopped using HP Sockets, the user can specify a run string with parameters in the PROCDEF file. Also, execution information such as the user identifier and password can be kept in this file.

File Definition File (FILEDEF). The FILEDEF file lists any attributes of files that will be transferred using HP Sockets. This file is optional. File attributes refer to the receiving node, and define such things as type (ASCII or binary), record size, and if the existing file can be replaced. Files that are not specified in this file can be moved using HP Sockets, provided the default file attributes used by HP Sockets are acceptable.

Data Definition File (DATADEF). This file defines the structure of the data transported between processes or applications. This file is optional.

Data Manipulation Definition File (DMANDEF). This file defines the manipulations that must be performed on the data to make it understandable to the receiving application. This file is also optional.

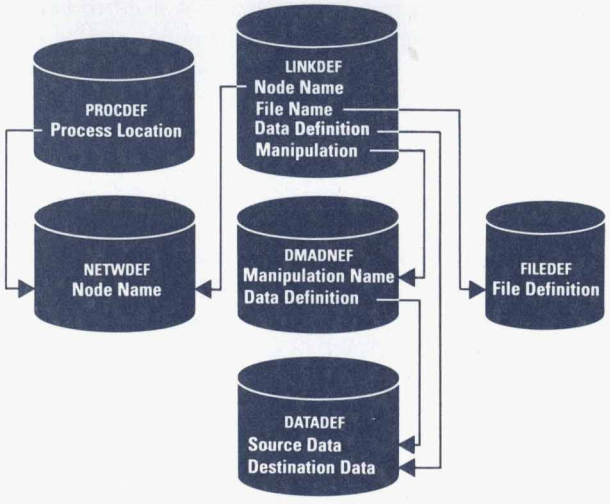


Fig. 1. HP Sockets configuration files and the fields in each file that associate one file with another.

```
#The Network Definition file, NETWDEF, must contain an entry for each node in the HP
#Sockets domain.
#
#This is the format for each entry. The numbers in parentheses indicate the maximum
#length of each field. Optional items are enclosed in square brackets.
#
#Node = LogicalNodeName : Machine Type
#                (16)                (16)
#[Startup = StartupStatus]
#                (1)
[CompilationNode = CompilationNodeFlag]
#                                (1)
#[MaxClones = MaxClonesNumber]
#
#[MaxMessageArea = MaxMsgAreaSize]
#
#[MaxPerProcess = MaxMsgAreaProcess]
#                                (3)
#NetworkType = Networking Type : PhysicalNodeName
#                                                (50)
Node = cpu1 : HP9000/S800
Startup = 1
CompilationNode = 1
MaxClones = 0
MaxMessagesArea = 100
MaxPerProcess = 50
NetworkType = LAN : hpiaclg.hp.com
#           Use the nodename command from the shell to get this name.
#
This is the end of the node definitions.
```

Fig. 2. A portion of a network definition file (NETWDEF).

Link Definition File (LINKDEF). This file ties the data manipulations in the DMANDEF file and the data definitions in the DATADEF file to the sending architecture, and to the sending and receiving languages. The same manipulation definition can be used in links describing differing languages and source system types. This modular approach to data movement allows the user to specify more easily many different types of links by leveraging from a single user message representation. This file is optional.

Link definitions are used to describe a specific link (data transfer) between two applications. Each link defines the language used by the source application to produce the data and the language that is used by the consuming application to access the data. Each link must also describe the format of the data at the sending and receiving ends of the link and any data manipulation that should be applied to the data.

In essence, a link definition name is used to bind the physical characteristics and operations that must be performed on data being transferred between two applications. This link name is used by the sending application at run time to invoke the operations necessary to get the data into a format usable by the receiving applica-

```
#The Process Definition file, PROCDEF, must contain an entry for each process known to the
#HPSockets Domain.
#This is the format for each entry. The numbers in parentheses indicate the maximum
#length of each field. Optional items are enclosed in square brackets.
#
#PLN = ProcessLogic Name : LogicalNodeName [:CloneFlag]
#                (16)                (16)        (1)
#[Exec =  ExecInfo]
#           (128)
#[Run = DefaultRunString]
#           (255)
PLN = lookupcp : cpu1 : 0
Exec = 100 user : passwd
Run = your_home_directory/tutorial/programs/lookupcp
#
PLN = inqcp : cpu1 : 0
Exec = 100 user : passwd
Run = your_home_directory/tutorial/programs/inqcp
#
#inqcp is the end of the process definitions.
```

Fig. 3. A portion of a process definition file (PROCDEF).

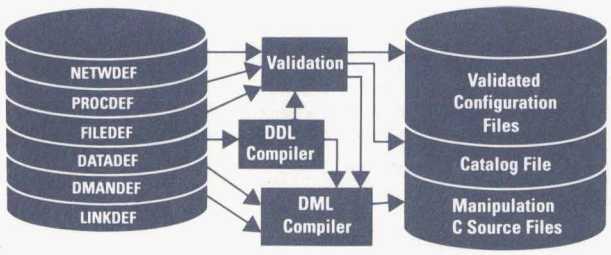


Fig. 4. The configuration validation process.

tion. At validation time, the link information is used to help generate the source code that carries out the operations at run time.

Examples of a data definition file, a data manipulation definition file, and a link definition file are provided in "Data Manipulation" on page 20.

Changing Configuration

Changing a configuration is a simple task performed via the HP Sockets manager. A configuration is changed to add nodes (computers) to the domain or remove them from it, or to alter definitions, processes, manipulations, or links. The steps required to change to a new configuration include:

- Modifying the configuration files. This can be done while HP Sockets is running a current configuration.
- Validating the new configuration files. This can be done while HP Sockets is running a current configuration.
- Shutting down HP Sockets when validation is successfully completed.
- Starting HP Sockets with the new configuration.

Configuration Validation

Before the configuration files can be used by the HP Sockets run-time modules they must be checked against one another for correctness, and the files containing the data definition and data manipulation definition data must be compiled. The process to get all this done is known as configuration validation (see Fig. 4). Configuration validation takes place on the administration node and includes the following activities:

- The validation program reads the network definition (NETWDEF), process definition (PROCDEF), and file definition (FILEDEF) configuration files from the user's directory.
- The validation program invokes the DDL (data definition language) and DML (data manipulation language) compilers, which read the DATADEF and DMANDEF files respectively. The DML compiler also reads the link definitions from the LINKDEF file.
- Code generation takes place in two phases. The first phase generates the code to perform data manipulations and occurs as each data manipulation is recognized. The second phase occurs after all of the data manipulation definitions have been processed and is responsible for generating the data manipulation C source files.
- The validation program generates a catalog file which contains information about the files used for the latest validation.

The validated configuration files are in a format that can be loaded into memory to become the memory-resident configuration tables.

For the code generation process described above, one area of great concern is the ability to add support quickly for new languages and computer architectures. To aid this goal, the code generator has been designed to be table-driven. Two principal types of tables are used. The first type describes the physical representation of the data types for a specific language on a specific architecture and the second is used to describe legal type conversions between data types and the functions to call to generate the code to perform the type conversions.

If communication fails because of a connection failure, the Onmc process notifies its child process Onmh (outgoing network manager helper) to begin trying to detect when it is possible to communicate with the node again. The Onmh, in turn, notifies the Onmc when to try to communicate with the node again. The Onmc sends the spooled messages when the cause of the network service failure disappears and the Onmc can successfully establish a connection to the node.

The Onmc spools messages destined for different nodes to the same spool file, rather than maintaining a spool file for each node. This is done primarily to save file descriptors for connections. The spool file is segmented into file blocks. A file block is associated with a node and all file blocks belonging to one node are linked together. Messages, including spooled messages, are always sent to a node in the same order in which they are received.

HP Sockets Management Daemon

The management of HP Sockets across various nodes linked by a LAN requires the existence of a daemon process on every node, which is responsible for performing the various management tasks on that node. This daemon is called the HP Sockets management daemon, or SMD. The attributes of the SMD include:

- Providing support for the HP Sockets functions on a node in the HP Sockets domain
- Remembering the steps of any task performed so that they can be undone if requested or if the network connection to the requester is lost
- Supporting a wide variety of requests from the HP Sockets manager (smain) on any node
- Providing extensiblity for future requirements
- Supporting requests from various nodes simultaneously.

The HP Sockets management daemon, as its name implies, is designed to run in the background and disassociate itself from the login session. It uses a slight variation of the approach suggested in reference 2.

The SMD is also responsible for scheduling the HP Sockets programs on the local node. For example, in the HP-UX environment some of the programs need to be scheduled as the HP Sockets user (hpskts) while others need to be scheduled as the root user. To accomplish this the SMD is itself scheduled as root. It then switches itself to being an hpskts user. This sets its real-user identifier and its effective-user identifier to hpskts and its saved-user identifier to root. So now the SMD can switch itself between an effective-user identifier of root and an effective-user identifier of hpskts as needed.

When the SMD is scheduled it also starts the error logger program and the file transfer daemon. It monitors these programs because they are critical to the operation of HP Sockets on that node. If either program fails then the SMD stops all activity on the node and shuts itself down. The SMD always keeps track of the current status of the node by saving status information in an internal variable called NodeState. This information can then be made available to any HP Sockets program that requests it.

Some of the requests that the SMD supports are mutually exclusive (e.g., the HP Sockets manager functions startup, shutdown, and switchadmin). Such requests are called major requests. Other requests like nodestat can be made at anytime on any node. To control the major requests on the node, the SMD maintains an internal flag called the Major-

RequestFlag. When an HP Sockets program wants to execute a major request, it first attempts to set this flag before proceeding with the request itself. Information about the major request currently being executed on the node is stored in a separate global variable called MajorRequest. The MajorRequest variable also keeps track of the connection on which this request is made. The SMD uses this connection and other factors to identify who has control of the MajorRequestFlag.

The SMD's primary task is to handle local or remote requests from the HP Sockets manager (see below). The SMD sets up a network call socket on which it can receive a connection request from the HP Sockets manager. After accepting the connection request, the SMD creates a virtual connection to the HP Sockets manager. This connection is left open until the HP Sockets manager decides to shut it down. If the connection is lost for any reason then the SMD reverses the steps taken during the request and clears the MajorRequestFlag (if it was set by the HP Sockets manager on the lost connection).

Once a connection has been established to the HP Sockets manager, the SMD processes the requests sent by the HP Sockets manager. The reply to the request will depend on the request itself and the success or failure of the SMD in executing the request. To facilitate the communication, the SMD has a well-defined message buffer that it transmits across the network. This buffer is used by all HP Sockets programs that want to communicate with the SMD.

Another function of the SMD is to help expedite certain requests, especially the startup of nodes across the LAN. One way is to have the SMD copy (i.e., pull) the files required by its node rather than have them copied (i.e., pushed) by the HP Sockets manager. This approach also gets around the need for the HP Sockets manager to have special permissions to do the push.

Fig. 9 shows a simplified illustration of the connections between the HP Sockets manager and its local SMD and some remote SMDs.

HP Sockets Manager

The HP Sockets manager program (smain) is the main administration component of HP Sockets. It can be run from any node in the HP Sockets domain. The HP Sockets manager is used by the system integrator for creating an integrated system and by the system administrator to manage the integrated system. The HP Sockets manager uses passwords to guard against unauthorized access.

The primary tasks that can be performed by a system administrator through the HP Sockets manager commands are:

- Starting up and shutting down the system from the administration node
- Establishing a security scheme for the system from the administration node
- Responding to errors recorded in the error log file by HP Sockets from any node
- Customizing the error log files to meet system needs from any node
- Checking status of the HP Sockets domain from any node

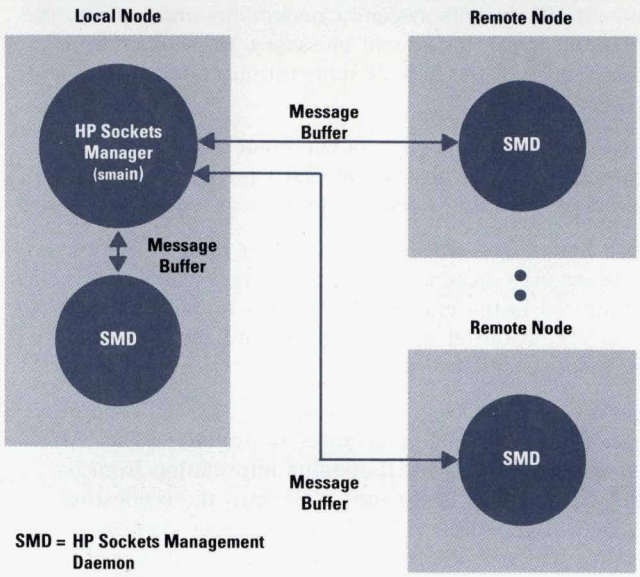


Fig. 9. Connections betwween the HP Sockets manager and its local SMD and remote SMDs. The message buffer contains status and request data.

- Switching administration capability from one node to another from an administration or alternate administration node
- Changing HP Sockets parameter values from the administration node only.

Switch Administration. The switch administration (switchadmin) function of the HP Sockets manager was created so that the major HP Sockets administration functions would not be lost if the administration node is unavailable. There can only be one administration node in an HP Sockets domain of nodes. However, any of the other nodes may be designated to be alternate administration nodes. Alternate administration nodes are the nodes in the HP Sockets domain that are able to take on the title and functions of the administration node. This is done using the switchadmin function on the alternate administration node.

When the user invokes the switchadmin function, the local HP Sockets management daemon (SMD) is contacted for status information. The SMD returns status about the local node, and a check is done to see whether the node requesting the switchadmin is an alternate administration node.

The switchadmin first tries to make a network connection with the SMD on the current administration node. It then sends a message to that SMD telling it to switch itself from an administration to an alternate administration node. This switch involves updating internal variables in the SMD that contain the administration status, and the logical and physical names of the new administration node. Next, the switchadmin module connects to its local SMD, sending a message telling it to switch itself to an administration node. This also involves updating the same SMD internal variables. Finally, after the old and new administration nodes are in synchronization, a list is created containing all the other nodes configured in the

current HP Sockets system. Connections are made to the SMDs on these nodes, and messages are sent informing these SMDs of the new administration node's logical and physical names.

If one or more nodes are down when a switchadmin is done, they can be resynchronized later by rerunning the switchadmin function from the administration node.

Node Status. The HP Sockets manager node status function (nodestat) gives information on one or all of the nodes configured in the current HP Sockets domain, or in another as yet unstarted domain whose configuration files are in a user-specified directory.

Nodestat makes a connection with the SMD on each node from which it is obtaining status information. The SMD on each node retrieves the status information from its internal variables, and sends it back to the requesting process. Typing nodestat at the HP Sockets manager prompt will give information about the local node, and typing nodestat <node logical name> gives information about a particular node. Typing nodestat * will list information about all nodes.

The information shown for node status includes:

- Logical name
- Physical name
- Run-time status (running or stopped)
- Administration class (administration, alternate administration, or nonadministration)
- Machine type
- Administration node logical name
- Administration node physical name.

Security. The HP Sockets manager security function allows the user to add and delete user names and passwords that are required to access the HP Sockets manager. The allowable commands for the security module are:

- Add. Add a user name and password to the user file
- Delete. Delete a user name and password from the user file
- List. List all user names in the user file.

The security file is disseminated to all the nodes in the domain at startup and after each update.

Performance in the HP Sockets Domain

Since it is impossible to predict the performance of HP Sockets for all possible configurations, some results are presented for several example test configurations. These tests represent typical configurations in which HP Sockets operates. Descriptions of factors contributing to performance results are also given.

Test Configuration

The system factors affecting HP Sockets performance include the CPU type, operating system, LAN activity, and network configuration. Tests were performed between HP 9000 Series 300 and Series 800 computers connected by a LAN. All systems used the HP-UX 7.0 operating system and contained 16 megabytes of main memory. All tests were run on an isolated LAN link. Except for virtual memory daemons, no other system processes were running during test execution. Fig. 1 shows three sending adapters and one destination adapter running on separate nodes. All messages were directed to a single destination adapter and both nodes were sending and receiving messages concurrently through HP Sockets. The message transit time between two adapters was estimated by measuring the time to transfer a large batch of messages (typically 5000 messages) between each sending and destination adapter pair. The message transit time was calculated as the time difference between the first send and the last receive, divided by the batch size. File transfer time was measured using the same method. Also, CPU utilization and the number of LAN packets going across the network were monitored while the tests were executing.

For the setup shown in Fig. 1, each sending adapter sent messages as fast as it could. The overall sending rate was increased by increasing the number of sending adapters. For stress tests, 96 sending adapters were successfully tested.

Performance Results

Some of the product-specific factors that affect the performance of HP Sockets are:

- The sending rate (in Fig. 1, the number of sending adapters)
- Whether delivery is with wait or without wait
- Whether delivery is guaranteed or not guaranteed
- The size of the message or file
- The data manipulation required for each message.

Message transfer times were studied for message sizes of 64, 256, 768, and 1024 bytes. HP Socket's maximum message size is 1024 bytes. The maximum file size tested was one megabyte.

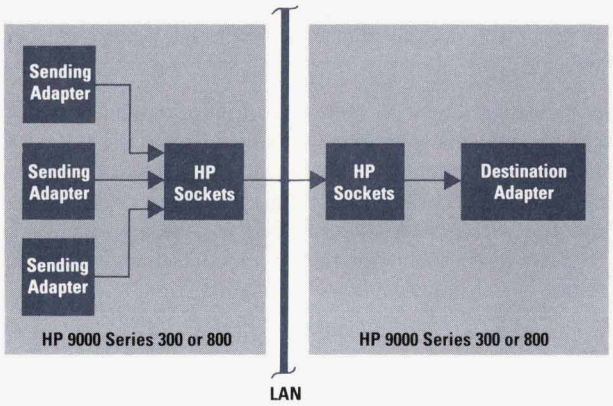


Fig. 1. Test configuration for performance tests.

A sending adapter can deliver a message with or without wait. When delivering with wait, the sending adapter does not continue until it receives an acknowledgment from the destination node. Therefore, the time for a round trip occurs between each message sent. In contrast, when a sending adapter delivers a batch of messages without wait, it can send messages at a higher rate.

The throughput is faster for a batch of messages delivered without wait. In fact, if the systems are not heavily loaded, delivery without wait can double the throughput. If the systems are heavily loaded, the increase in throughput may not be noticeable.

Similarly, a sending adapter can deliver a message with or without guarantee. Messages delivered with guarantee are written and flushed to a disk file; thus two disc accesses (a write followed by a read) are done for each guaranteed message transfer. In contrast, messages delivered without guarantee are kept in shared memory. Guaranteed message delivery requires more system resources (CPU and disk I/O). However, the total message transit time may be unaffected, unless the systems have extremely high I/O activity.

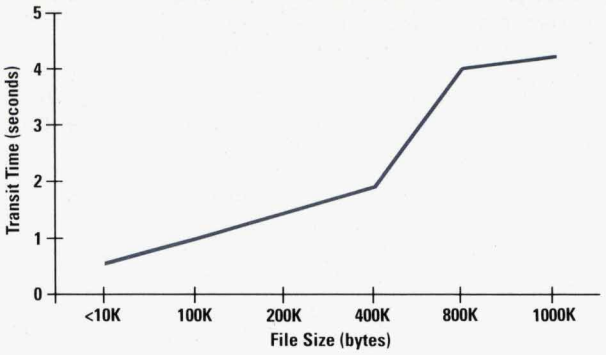


Fig. 2. HP Sockets file transit time as a function of file size using an HP 9000 Series 370.

Fig. 2 shows the performance for file transfer between two HP 9000 Series 370 computers. File transfer time for a small file (less than 10 kilobytes) is about 600 milliseconds. File transfer time increases with file size—from about 0.6 second for a 10-kilobyte file to about 4 seconds for a one-megabyte file.

Fig. 3 shows the message transfer time as a function of message size. Again two HP 9000 Series 370 computers were used. The data in Fig. 3 was generated for delivery without wait and without guarantee, but tests with wait and with guarantee showed results that were only slightly higher. The line labeled "No Data Manipulation" shows the results for messages sent without data manipulation. Since these messages were transmitted directly to their destination, the transfer time was fast and showed little dependence on the message size. Even for the maximum-sized messages, transfer time without data manipulation was always less than 22 milliseconds.

The line labeled "Host Data Converted" is for messages that required simple data manipulation. Message transfer time increased with the message size to about 70 milliseconds for large messages. The line labeled "Extensive Data Manipulation" is for messages transferred with extensive data conversion. In these tests, an array with elements of type integer was converted to an array with elements of type real. For example, a message size of 1024 bytes required 256 integer-to-real conversions. Under these conditions, transit time was always under 100 milliseconds.

Performance tests for HP Sockets are conducted continually under many environments and conditions. The results given here represent a typical environment, consisting of a single sending adapter and a single destination adapter, each on separate nodes, sending a large batch of messages. If CPU and/or LAN loading is normal, these results represent a conservative estimate.

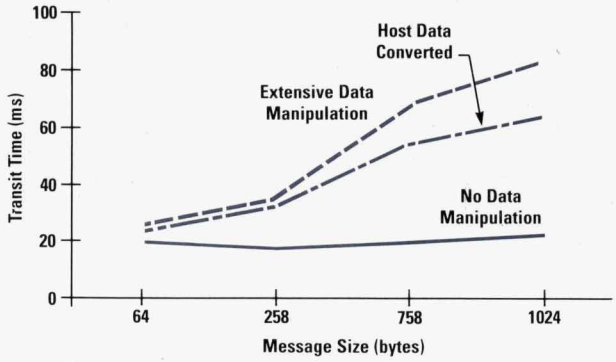


Fig. 3. HP Sockets message transit time as a function of message size using two HP 9000 Series 370s.

Startup and Shutdown

The startup of the HP Sockets run-time programs on the various nodes connected by a LAN poses some interesting problems. It is necessary to distribute the configuration files to all the nodes in the network that need them and to coordinate the multistep startup sequence concurrently on all the nodes. At the same time, the startup sequence must be flexible enough to allow the user to back out of the startup sequence at any time. A further requirement is the need to abort the startup automatically if a major catastrophe occurs, like the loss of the connection between the startup program and the local SMD. Additionally, the number of nodes that need to be started can far exceed the system limits on the number of open connections that any process, like the HP Sockets manager, is allowed to have. This section outlines how these problems were solved and Fig. 10 shows some of the data flows between the administration node and a node involved in the startup process.

All network connections are established using a connection-oriented protocol, such as TCP/IP.[3] Once a connection has been established, the HP Sockets manager on the administration node, with the help of the local and remote SMDs, keeps the connection open until all the steps in the startup sequence are completed. The design of the SMD allows the HP Sockets manager to request that the connection to a particular node be temporarily relinquished, thereby suspending the startup process on that node before the sequence is finished. The SMD can also automatically undo the startup sequence on a node if the network connection between the SMD and the HP Sockets manager is lost before the entire sequence is done.

Before beginning a startup sequence, six configuration files must be validated (see "Configuration Files" on page 13). Configuration file validation converts the files to a format that allows them to be loaded into memory. Once the configuration files have been validated, the user invokes the startup command from the HP Sockets manager screen, and specifies (among other things) the directory path to the validated configuration files to be used for the startup.*

The HP Sockets manager connects to the SMD on the local node and sets the MajorRequestFlag maintained by the SMD to ensure that the node is not interrupted during the startup sequence. This network connection to the local SMD is never relinquished for any reason. If the connection is lost then the startup sequence is aborted on all the remaining nodes and an automatic cleanup is performed by the SMD on each affected node.

After setting the MajorRequestFlag, the HP Sockets manager requests the local SMD to report on the current status of the local node. The HP Sockets manager then verifies that the startup request was issued from the administration node, and if not, the startup sequence aborts.

*There can be more than one set of validated configuration files, each in a different directory.

If the user has specified a new configuration file directory in the startup call, the HP Sockets manager copies the configuration source files and the validated configuration files to its working directory (① Fig. 10). The HP Sockets manager then requests the SMD to load a copy of the validated configuration files into local memory, creating the run-time configuration table (② in Fig. 10). The startup program extracts configuration information from this table to build a singly-linked list (called the domain node list) containing all the nodes that are configured in the HP Sockets domain (③ in Fig. 10). This list enables the startup program to coordinate the network-wide startup of all the nodes by keeping track of each node's status. While the domain node list is being built, a list of compilation nodes is also created.

Compilation nodes are nodes in the HP Sockets domain that deliver and create (via C compiles) data manipulation modules for specific machine architecture groups. For example, one of the HP 9000 Series 300 machines in an HP Sockets domain would be designated for compiling and distributing data manipulation modules to other Series 300 machines in the domain. The same would be true for each of the other machine architecture groups (HP 9000 Series 800, HP 3000 Series 900) in the domain. See "Data Manipulation" on page 20 for more about data manipulation modules.

After the lists are built, the HP Sockets manager connects to the SMD on each node that has been designated by the user for startup. If it cannot establish a connection to a node in three tries, startup on that node is aborted and the domain node list is updated. The connections are all established in parallel. When the HP Sockets manager runs out of nodes to be connected to or cannot make any more network connections (because of system or process limits) then it waits for responses to come back on the connections that have been established.

To reduce the time required to start up all the nodes in the domain, files are copied (pulled) by the SMDs on the respective nodes.

The HP Sockets manager uses a connection management table to keep track of all the remote SMDs to which it is connected at any time. The table is an array of pointers to the corresponding nodes in the domain node list. The status of the network connection from the HP Sockets manager to the SMD on a remote node is kept in the appropriate list element. For example, whenever the startup process is aborted on a node (for whatever reason) the domain node list is updated accordingly.

As a response is received on a connection, the HP Sockets manager processes that response and posts the next request (in the startup sequence) to that SMD. If there are nodes that did not get a chance to perform the current step in the startup sequence because there were no more network connections available to the HP Sockets manager at that time, then the node that just replied is temporarily disconnected from the HP Sockets manager. The HP Sockets manager then connects to a node that is behind in the startup sequence and posts the current request. This approach is followed for each step in the startup sequence. Only the connection to the local SMD is never relinquished. If there are no nodes that are behind in the startup sequence then the next request in the sequence is posted to the node that just replied.

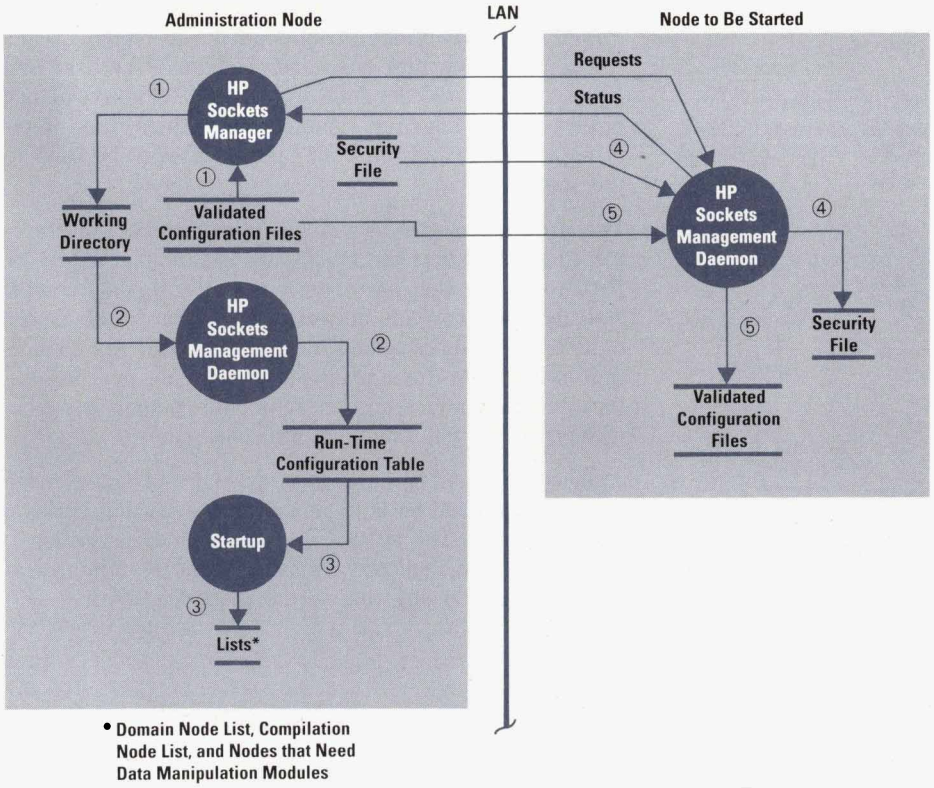


Fig. 10. Some of the processes and data flows involved in HP Sockets startup.

The HP Sockets manager asks the SMD to set the MajorRequestFlag on that node. (If the flag cannot be set then the startup process is aborted on that node.) After setting the MajorRequestFlag, the HP Sockets manager asks the SMD to report on the current status of the node. The startup sequence will be halted on a node if the machine type does not match the machine type configured, or the run-time programs are running but with a different configuration identifier from that used in the startup.

The HP Sockets manager checks to see if any data manipulation modules need to be built for any of the machine architecture groups in the configuration. If a data manipulation module needs to be built for a given machine architecture group then the HP Sockets manager connects to the SMD on the compilation node for that machine architecture group. It requests that the SMD schedule the data manipulation module builder. The HP Sockets manager then communicates directly with the data manipulation module builder and requests it to pull the relevant C source file from the administration node (① in Fig. 11). Then a list of nodes (of that machine architecture group) that require data manipulation modules is supplied to the data manipulation module builder (② in Fig. 11). For each node that needs a manipulation module, the builder:

- Compiles the appropriate C source files (③ in Fig. 11)
- Builds the data manipulation module
- Distributes the data manipulation module to the node by requesting that the SMD on that node pull it over (④ in Fig. 11).

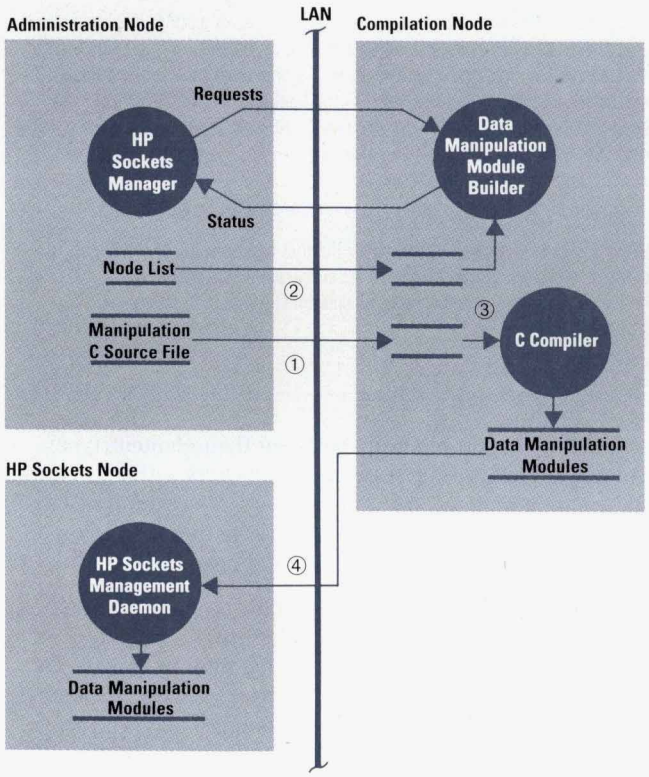


Fig. 11. Activities taking place between the administration node and a compilation node during startup and the creation of data manipulation modules.

As the data manipulation module builder completes the task for a given node on the list it reports the status back to the HP Sockets manager. If the data manipulation module cannot be built for a given node then that node is not started.

After finishing with the compilation nodes, the HP Sockets manager asks the SMDs on the other nodes to pull the validated configuration files (⑤ in Fig. 10) required to build the memory-resident run-time configuration table on that node. Once this step has been successfully accomplished the SMD is asked to prepare the run-time programs on that node for startup. All these steps are performed on each node in the sequence indicated but are done in parallel on all the nodes.

Once the run-time programs have been prepared for startup on all the nodes designated by the user (and did not fail any of the steps outlined above), the user is asked if the startup should proceed or be aborted. The user's decision is communicated to the SMD daemon on all the nodes waiting to start.

When these steps in the startup procedure are completed, the SMD on each node is told to transfer (i.e., pull) the security file from the administration node (④ in Fig. 10) so that all nodes in the domain have the same security file.

The user is allowed to abort the startup process at any time except when the configuration files are being transferred from the user's directory to the HP Sockets working directory and when the data manipulation modules are being built on the various compilation nodes. If the HP Sockets manager decides to abort the startup sequence on any node for whatever reason, then the steps that were accomplished so far on that node are undone by the SMD on that node and the status of the node is returned to its prestartup state.

If the node that is being started already has the latest configuration files available from a previous startup, then some of the steps listed above are omitted. This significantly speeds up the startup process.

The user can shut down HP Sockets from the HP Sockets manager screen on the administration node. During shutdown, the HP Sockets manager uses an approach that is very similar to that used during startup. It goes through the same sequence of first setting the MajorRequest Flag held by the SMD on the local node. After checking the status of the local node, it connects to the SMDs on all the nodes that need to be shut down.

Once the SMDs on all the nodes marked for shutdown have been alerted to the impending shutdown, the user is asked if the shutdown should proceed or be aborted. The user's decision is communicated to the SMD on all the

HP Sockets Gateway

The HP Sockets Gateway product uses a client/server model to extend HP Sockets capabilities to machines not using the HP-UX or MPE XL operating systems. The client is a user application running on a machine such as an IBM 3090 mainframe or a PC. The server is an HP Sockets adapter (called a server adapter) running on a gateway node, which is an HP-UX node within the HP Sockets domain that has been designated to run the gateway server.

The gateway server maintains a network connection to one or more client applications running on the client machine. Every gateway node has one server daemon for each networking service; it creates server adapters required by the client applications. The server adapter provides the functionality of HP Sockets to a client application. It is also called a virtual adapter since it acts on behalf of a client application.

An application on a client machine is linked with the client HP Sockets access routine library. This library lets a client application send or receive message or file data to or from any other processes within the HP Sockets domain. The parameter list and return values for this library are the same as for the HP Sockets HP-UX access routine library. Functional differences between the client access routine library and the HP Sockets HP-UX access routine library have been minimized.

The current release of the HP Sockets Gateway uses TCP/IP for interprocess communication across the network and ftp (file transfer protocol) for file transfers. Currently software is available for connections to an IBM 3090 mainframe running the MVS operating system, or a PC running MS-DOS.

HP Sockets Gateway Components

The HP Sockets Gateway components consist of the server daemon, the server adapters, the client HP Sockets access routine library, and the client applications. Fig. 1 shows the server daemon and three server adapters running on the gateway node, MachineA. The dotted lines in the figure indicate that the server adapter processes are child processes of the server daemon process. Each of the three server adapter processes services a client application. The node labeled MachineB could be a PC running MS-DOS. The node labeled MachineC could be a mainframe like the IBM 3090 system.

The purpose of the server daemon process is to accept connection requests and create server adapter processes. The server daemon is always active, ready to create multiple server adapters when connection requests are made. It listens at a well-known port number associated with the HP Sockets Gateway service. The client applications make connection requests to the server daemon through the gateway service via access routine library calls.

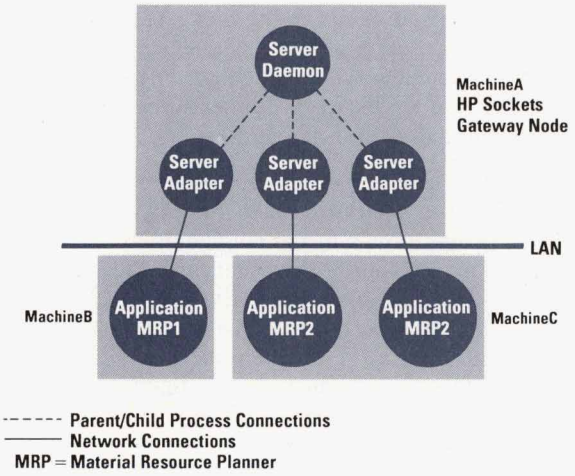


Fig. 1. A server daemon and three server adapters running on the gateway node (MachineA).

Each server adapter is associated with an HP Sockets logical process name. Since a server adapter is a virtual adapter, this logical process name is also the effective logical process name of its client. The logical process names are entered in the HP Sockets process definition file (PROCDEF), described in "Configuration Files" on page 13.

All access routine library calls are serviced by the server adapter on behalf of the client application. For example, when a client calls the access routine SpReadQ, a message is sent over the gateway network connection to the server adapter. The message contains all the parameters of the SpReadQ access routine call. The server adapter then issues an SpReadQ call to HP Sockets using the same parameters. When the SpReadQ call completes at the server adapter, it returns the output parameters and return codes to the client application. Except for the SpInit routine, all calls to client library access routines are handled the same way.

When a client issues a call to SpInit for the first time, a new server adapter is created. When the SpInit call completes successfully, the client has a virtual connection to a dedicated server adapter, and through it to HP Sockets.

nodes waiting to shut down. The user is allowed to abort the shutdown process at any time.

Using a single control point for startup and shutdown of the entire HP Sockets domain relieves the user of the obligation to start and stop each communicating CPU individually. In addition, HP Sockets automatically distributes all control information to designated alternate administration nodes to prevent the single control point from being a single point of failure. This prevents undelivered messages from being lost or processes from hanging on unfulfillable reads.

Data Manipulation

The data produced by one program may not be usable by a different program because the selection of elements in that data is not what the receiving program expects. Less obvious is the problem of the basic data element types not being readable by a receiving program either because the sending and receiving programs are written in different programming languages or because they reside on machines that have different architectures.

The same data type passed from one program to another can have a different format in each program if the programs were written in different programming languages. The same data type sent and received between two programs written in the same programming language can have different formats if the two programs were compiled on different machine architectures. Not only can the data type formatting be different when programs reside on different machine types, but the data type sizes and alignments (address locations used in memory) can also be different between the two programs. Most of these differences are because the language compiler optimizes the code by taking advantage of the data type efficiencies

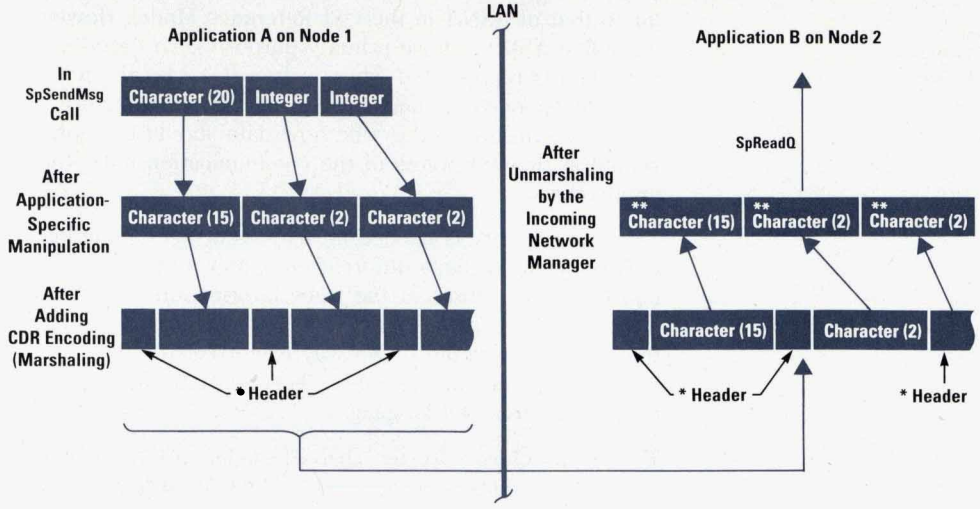


Fig. 12. A simplified illustration of the data manipulation, marshaling, and unmarshaling of data in the HP Sockets domain.

and sizing that the underlying machine architecture provides.

A brute-force method of dealing with data type format and language-to-machine incompatibilities is to write special conversion routines converting data types from language A on machine type B to language C on machine type D. These conversion routines would grow in number and complexity with the addition of each new language or machine type supported. The amount of conversion code would eventually become unwieldy and difficult to maintain. Another drawback is that one would have to go back and add code to already supported machines to support the new language or machine type.

The HP Sockets solution to this problem is twofold. First, HP Sockets uses a standard, common way of representing values for a particular data type independent of the source or target language and machine type. This is called common data representation, or CDR. Secondly, the application-specific manipulation is done only on the sending machines. Fig. 12 provides a simplified illustration of what happens when a data structure from application A is manipulated to be compatible with the data structure required by application B, which is on another node. Note that CDR conversion is done on both nodes, but application-specific manipulation is performed only on node A, the sending node.

Common Data Representation

Implementation of the common data representation format on any system in the HP Sockets domain requires two sets of conversion routines for each language supported for that machine type: one set to convert from one local language into CDR and another to convert from CDR back into that local language. Going from a local language to CDR is called marshaling and going from CDR to a local language is called unmarshaling. A big advantage of this method is that as new languages and machine types are added to the network, code changes to support the new languages or machine types are isolated.

The Basic Encoding Rules (BER) of Abstract Syntax Notation One (ASN.1)[4, 5] were chosen for the format of the CDR data types. ASN.1 is an OSI (Open Systems Interconnection) standard that specifies the different types of data structures that can be transferred between protocol layers of an OSI stack. The Basic Encoding Rules define the encoding and decoding rules for these data structures.

As shown in Fig. 12, the target information is marshaled and sent with the CDR data so that the receiving side can unmarshal and localize the CDR data appropriately. Thus, the CDR data is self-describing for eventual unmarshaling.

ASN.1 does not make the CDR data completely self-describing. For example, ASN.1 allows marshaling an integer into a generic representation. That integer will eventually be unmarshaled into a localized integer data type. The problem is in determining which local integer data type to convert the CDR data into. If the local language is C, the CDR integer could be converted into either a short integer, integer, or long integer data type. The target data type to use is specified by the user through data definition declarations made via the data definition and data manipulation languages, or DDL and DML respectively. These languages make up the application-specific manipulations shown in Fig. 12.

This extra, self-descriptive information is called type attributes. For example, the ASN.1 string type encoding indicates the current size of the string but not the maximum size it should be on the receiving end of the transfer. This maximum size is included with the CDR data and is a type attribute.

Because of the complexity of the task of data manipulation and the need for the best performance possible, there is tight coupling between the CDR marshaling and unmarshaling routines and the HP Sockets-generated data manipulation code that calls them. Thus, the CDR routines cannot be used apart from the rest of the product.

```
struct{
    char Operator[20];
    int Month;
    int Day;
    int Year;
}id;
```

(a)

```
struct{
    char Operator[15];
    char Date[11];
    }r1;
```

(b)

```
DATA_DEFINITION
BEGIN

/*Define structure id as a record named
input_data in DDL*/

input_data = RECORD OF[
Operator        :  STRING[20]
Month           :  INTEGER;
Day             :  INTEGER;
Year            :  INTEGER;
];

/* Define structure r1 as a record named report1 in
DDL*/

report1 = RECORD OF[
Operator        :  STRING[15];
Month           :  ARRAY[2] OF CHAR;
Slash1          :  CHAR;
Day             :  ARRAY[2] OF CHAR;
Slash2          :  CHAR;
Year            :  STRING[5];
];
```

(c)

```
/* The format of the Data Manipulation Definition file (DMANDEF) is shown below.

DEFINE_MANIPULATION ManipulationName;
    SOURCE_DATA : SourceDataStructureName;
    DESTINATION_DATA : DestinationDataStructureName;

BEGIN_MANIPULATION
<manipulation statements>;
END MANIPULATION;
*/

DEFINE_MANIPULATION i_to_report1;
    SOURCE_DATA         :    input_data;
    DESTINATION_DATA    :    report1;

BEGIN_MANIPULATION
    MOVE input_data TO report1;
    report1.Slash1='/';
    report1.Slash2='/';
END_MANIPULATION;
```

(d)

Fig. 13. An example of defining a numeric-to-ASCII conversion in DDL. (a) Original structure in some sending application. (b) Structure in which the operator and date information will exist in the receiving application.(c) DDL definition for the two structures. (d) Data manipulation definition for moving the data in a data structure like input_data to the structure defined in report1.

Data Definition Language

Data definition means defining the format of the data transmitted between processes or applications. HP Sockets uses these data definitions along with the data manipulations defined via the data manipulation language to manipulate the source data so that its format is understandable by the destination process.

These data definitions for HP Sockets are written in the data definition language (DDL), which is a high-level language for specifying the format of the data exchanged by processes. The DDL is used to describe the type and structure (logical layout) of data produced and consumed by applications. A data definition is independent of any particular computer language's or machine architecture's physical representation.

The DDL serves the function of a presentation layer similar to that of ASN.1 in the OSI Reference Model. However, unlike ASN.1, whose primary purpose is to describe how data is represented while in transit across the network independent of language or machine architecture, the DDL is meant to describe how data should be represented at the end points of the communication link—the applications.

This functionality is needed in heterogeneous computing environments because different computer architectures and languages represent the same information in different ways (see Figs. 13a and 13b). Without a description of the data, there would be no way to correct the data for the representational differences between differing computer architectures and languages.

The syntax chosen for the DDL is similar to that of other high-level languages, such as C or PASCAL. A lex/yacc-based compiler is used to compile the DDL source file and to produce an in-memory symbol table. This symbol table is used later to drive the code generator which produces the C source files that perform data manipulation and conversion to a common data representation. Fig. 13c shows the DDL declarations required to describe the structures defined in Figs. 13a and 13b.

Data Manipulation Language

As mentioned earlier, data types and data organization may differ between source and destination nodes. Therefore, the manipulations that must be performed on source data so that it becomes the destination data must be defined. Manipulations require the definition of the source and destination data contained in the data definition file.

Data manipulations are described using a high-level language called the data manipulation language (DML). This language allows the systems integrator to specify how to transform the data produced by one application so that it is more acceptable to another application. Standard transformations include the ability to reorder the fields in a data structure, delete data fields that are not needed by the consuming application, and add and initialize new data fields not provided by the source application but needed by the destination application. Fig. 13d shows the DML statements that define how to move and manipulate data coming from a data structure like that defined in the input_data record shown in Fig. 13c to the report1 record shown in Fig. 13c.

In addition to manipulating the structure of the data, capabilities have been provided for automatic type conversion and resizing of data fields. For example, converting an integer into an ASCII string, an integer to real value, or a short integer into a long integer.

As with the DDL, the DML source file is compiled using a lex/yacc-based compiler. However, code generation is postponed until a data manipulation has been completely recognized and checked for semantic errors.

Data Manipulation Module

The DDL and DML specifications created by the user wind up in the user-defined configuration files.The user also indicates the programming language of the sending and receiving applications for each manipulation or data definition in the link definition file (see Fig. 14). This information enables HP Sockets to provide automatic conversion between different languages—for example, converting a FORTRAN two-dimensional array (stored in column-major order) into a C two-dimensional array (stored in row-major order), or a C string into a Pascal string. The languages currently supported in HP Sockets are C, FORTRAN, Pascal, and COBOL. Before HP Sockets is started up, these files are compiled via DDL and DML compilers into C source files and then the C files are compiled into executables called data manipulation modules. These data manipulation modules are placed on the machines where they will be used. Configuration files and the creation of data manipulation modules are described on page 13.

The data manipulation module eliminates the need for a sending application to be knowledgeable about the data representation required by the destination application.

Fig. 15a shows the values assigned to a data structure like that shown in Fig. 13a by a sending application, and

```
#The Link Definition file, LINKDEF, contains HP Sockets link definitions.
#
#This is the format for each entry. The numbers in parentheses indicate the maximum
#length of each field. Optional items are enclosed in square brackets.
#
Link = LinkName:M = ManipName | D = DataDefName | NULL
#          (16)              (16)              (16)
#[SourceLanguage = SourceLanguage]
#
#[SourceFileDef = SourceFileDefName | DEFAULT]
#                                (16)
#[DestLanguage = DestLanguage]
#
#DestFileDef=DestFileDefName | DEFAULT]
#                          (16)
#SourceName=SourceNodeName[:SourceNodeName][SourceNodeName]...[\]
#                          (16)              (16)              (16)
Link =  link1:M = i_to_report1
SourceLanguage = C
DestLanguage = C
SourceName = cpu1
#
#This is the end of the link definitions.
```

Fig. 14. Link definition file for the manipulation described in Fig. 13d.

Application Call	Output

strcpy(id.Operator, "Barry Littlename";
id.Month = 1;
idDay = 30;
id.Year = 1990
	Operator is Barry Littlenam
	Date is 1/30/1990

(a)	(b)

Fig. 15. (a) Values assigned to the original data structure shown in Fig. 2a. (b) After manipulation and printout from the destination application.

Fig. 15b shows the printout from the receiving application after data manipulation.

Acknowledgments

The authors would like to acknowledge the HP-UX team members—Thong Pham, John Frohlich, Daryl Gaumer, Pam Munsch, Le Hong, Mari Budnick, Sharat Israni, and Clemen Jue—for their effort and dedication toward the completion of the HP Sockets product. We would also like to acknowledge Diane Ho, Tim Tsao, Larry Woods, and Lee Yu for their significant contributions in porting HP Sockets from the HP-UX operating system to the MPE XL operating system. Marvin Watkins and David Wathen made the foreign host connections possible. We would also like to thank the marketing department, especially Dan Kaplan, Charlie Tuan, Joe Bac, and Kathleen Lowe, for making this product a reality. In addition, Kent Garliepp helped to make this article possible.

References

1. K. J. Faulkner, et al., "Network Services and Transport for the HP 3000 Computer," *Hewlett-Packard Journal*, Vol. 37, no. 10, October 1986, p. 15.
2. D. Lennart, "How to write Unix daemons", *UnixWorld*, December 1988, pp. 107-117.
3. K.J. Faulkner, et al., *op. cit.*, p. 14.
4. *Information Processing Systems - Open Systems Interconnection - Specification of Basic Encoding Rules for Abstract Syntax Notation One (ASN.1)*, ISO 8825: 1987 (E).
5. K. K. Banker and M. A. Ellis, "The Upper Layers of the HP OSI Express Card Stack," *Hewlett-Packard Journal*, Vol. 41, no. 1, February 1990, pp. 28-36.

Rigorous Software Engineering: A Method for Preventing Software Defects

Formal specification languages enable software engineers to apply the rigorous concepts of discrete mathematics to the software development process.

by Stephen P. Bear and Tony W. Rush

The technology base of electronics and computer companies like Hewlett-Packard is changing—software has become pervasive. In all market sectors the proportion of software in individual products continues to increase. Fig. 1 shows the dramatic increase of software in products of one family developed between 1979 and 1989.

Product A produced in 1979 was entirely hardware, and contained no software at all. Product B contained 38 KNCSS (thousands of lines of noncomment source statements) of Pascal. In 1986, product C contained 200 KNCSS of structured assembly language, which is equivalent to about 100 KNCSS of a high-level language like Pascal or C. The current member of the family, product D released in 1989, contains some 350 KNCSS of C.

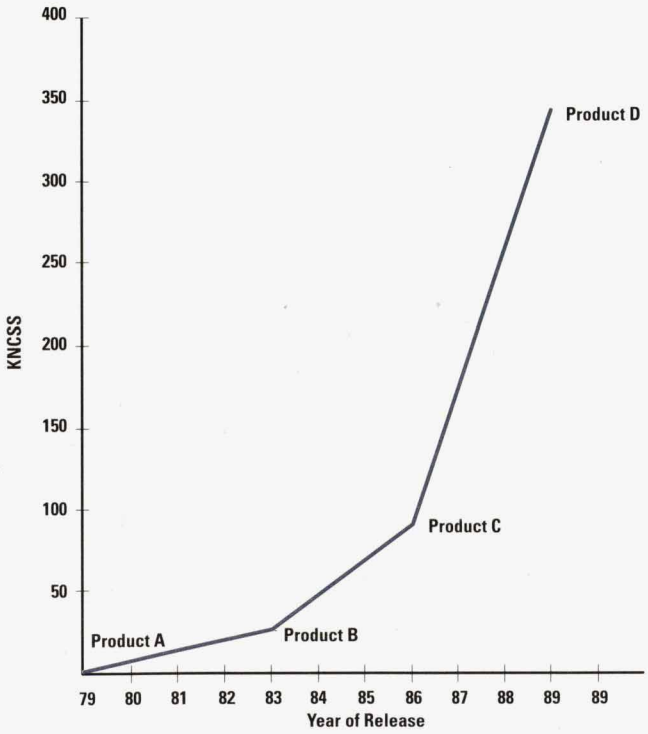


Fig. 1. Software content of a particular medical product family from 1979 to 1989.

Growth like this is not accidental because there is a feedback loop that drives the increased functionality provided by increased software. Software enables us to design products with greater functionality. Initially, this functionality provides a competitive advantage, but it soon becomes essential to success in the market. To make an impact, each new product must have more features and more software.

Software Takes Too Long

This massive change in products, with software providing much of the functionality, is happening despite significant shortcomings in the current software development processes. The principal problem is that software development takes too long. The development process is difficult to control and defect removal can introduce seemingly arbitrary delays.

Delays increase development costs, but more important, they also reduce market share and affect overall lifetime profit. Studies of high-growth, short-lifecycle markets suggest that a six-month delay in introducing a product can reduce profit by 33%.[1] One HP Division estimates that for each new product, finding and removing bugs can cost over 1 million dollars.[2]

Software development takes too long because it is wasteful. Errors and defects are introduced during development, but are not detected quickly. Further work is then built upon the erroneous development. When a defect is detected, there are many interlocking details, and these must be reworked or discarded. During development this happens repeatedly. Again and again, work carried out at one stage is thrown away and must be replaced.

Defects not detected until late in the software development process can affect large amounts of work. They are difficult, time-consuming, and expensive to correct. Defects detected early in the development process require less work and typically they are easier and cheaper to correct.

Fig. 2 shows the cost of removing defects detected at various points in the software development process, as calculated by the software quality engineering group at

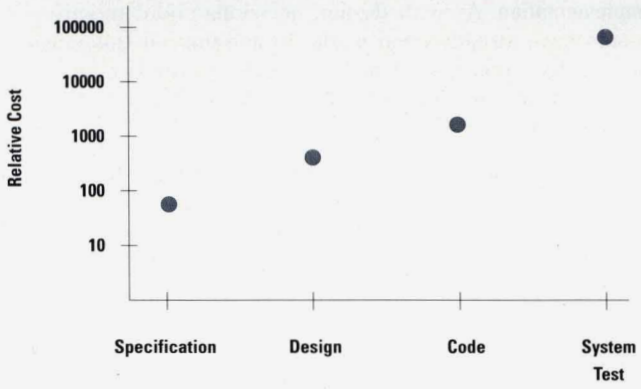


Fig. 2. Calculated cost of defect correction at point of detection in the lifecycle.

one HP Division. Notice that costs of removal increase exponentially as the development proceeds.

It is instructive to look at these figures in a slightly different way, and to consider the cost of introducing an error at various points of the development process. If an error is detected by a code inspection or design review soon after it is introduced, it can be fixed quickly and cheaply. For example, if a specification error is detected at specification time, it will be cheap to repair. However, if it is not detected at this stage, then it is likely to lie dormant until the consequences of the error are observed, by which time it will be much more expensive to correct.

It turns out that the introduction and detection of the consequences of an error are nested because typically:

- Errors introduced during low-level design and coding are observed quickly during compilation and unit test
- Errors introduced during high-level design are observed during integration and system test
- Errors introduced during specification are not observed until system test and use.[3]

We can use this relationship to estimate the relative costs of introducing a defect at various points in the development process. It shows that a defect introduced early in the development process—and not detected immediately—will be much more expensive to fix than an error introduced during coding (see Fig. 3).

Clearly, defect prevention during specification and design is extremely important. Getting it right at this point is where it really matters. However, if we look at current practice, we see that the analytical and descriptive tools used during specification and design are very weak. There is little attempt to capture behavior at the specification stage. Early narrative documents describe features, but these are inevitably ambiguous, incomplete, and often contradictory. High-level descriptions of behavior are just as vague, and precise descriptions often resort to implementation details.

The result is a rush to code. Developers do not spend time understanding and describing behavior because it is hard to capture and communicate. Instead they use the implementation to document issues and resolve problems. Often, the implementation is the first precise description of what the software will do.

Many important decisions about overall behavior are made by programmers working at the lowest level of detail. This is not a good way to think about behavior, and decisions made in this way are often inconsistent or undesirable. Aspects of behavior emerge from interactions that have never been explicitly considered or analyzed. Behavioral defects embedded in the code are difficult to detect by inspection or review. However, they are the defects that begin the costly cycle of rework and waste.

Rigorous Software Engineering

Rigorous software engineering is an approach to software development that addresses quality and productivity by emphasizing the early stages in the development process. Rigorous software engineering concentrates on developing an early, precise understanding of the required behavior of the system under development. Expensive specification and design errors can be avoided, or detected and corrected before the implementation begins.

The rigorous software engineering philosophy could be summarized as one of defect prevention. Think carefully about what you want to do and get it right the first time.

The approach is to develop an abstract, but precise description of the behavior of the software system. This description can be reviewed to ensure that the system does what is required. If there are problems, they can be fixed quickly and cheaply—long before an implementation is created.

Underlying the rigorous approach are formal specification languages. These are mathematically based languages that provide support for abstract and precise descriptions of software systems.

Rigor in the Software Lifecycle

Rigor can be used effectively at all stages of the software lifecycle. Naturally, if the early stages of the development are more error-prone or more costly if flawed, then those stages are likely to benefit most.

If we take a simple model of software development—specification then design then implementation then testing—we can show the application and benefits of

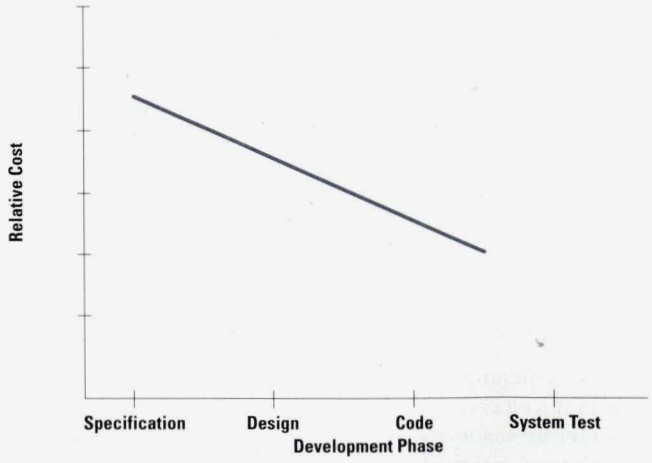


Fig. 3. Relative costs of defects introduced in various development phases.

rigorous software engineering for each stage. In real software development, these phases will appear, but they may be interleaved, repeated, or omitted.

Specification. Rigorous techniques are particularly useful at the specification stage. A rigorous system of specification is both abstract and precise. Abstraction focuses attention on the essential aspects of behavior. An abstract description is not cluttered or confused with irrelevant details. It makes the behavior of the system easier to understand and to think about. Precision encourages careful thought because issues must be resolved and cannot be hidden in an ambiguous narrative.

Together, abstraction and precision make it possible to communicate the proposed behavior. Reviews and inspections can be more effective. Other people can understand what is suggested because they can analyze consequences and identify problems or omissions.

The specification of the system will construct a model that shows all the required properties of the system. This model can be used to resolve difficult issues of behavior. If the model cannot resolve these issues, then it is flawed and needs improvement. In practice, this makes it difficult to ignore tricky areas of the specification.

A rigorous specification, then, provides benefits to both the authors of the specification documents and the reviewers of those documents. Clarity of thought and concept helps prevent defects from being introduced. Clarity of expression empowers reviewers and helps ensure that defects are quickly identified and removed.

Design. A feature of the rigorous software engineering approach is the ability to separate concerns. In the specification phase, emphasis is placed on what a system is to do. In the design phase emphasis is placed on how the system is to be built because decisions about behavior have already been made. It is not necessary to resolve requirements issues while doing design.

Precise specifications provide detailed guidance for designers. Teams of designers know what is to be done. This makes it easier for the development to be well-managed because tasks can be defined precisely (design the feature whose specification is as follows) and allocated to developers. Misunderstandings causing integration problems or system errors can be dramatically reduced.

It is straightforward to combine the rigorous approach with techniques such as structured design.[4,5] This gives many benefits, especially the ability to trace which parts of the design implement which parts of the specification. This traceability is an important attribute of any development process, allowing subsequent enhancements to code to be done in a controlled way and improving the quality of the finished software product. The article on page 51 shows the combination of structured design techniques and rigorous specifications.

Good specifications are especially important when software is developed by subcontractors. Rigorous specifications can be made tight enough to allow subcontractors to implement precisely the desired functionality, and for there to be much less disagreement over the required behavior of the system.

Implementation. As with design, decisions about requirements have already been made. In addition, at this stage, design decisions have also been made. Effort is concentrated on producing an efficient implementation of the desired behavior.

The code modules can be traced to the design descriptions and the design descriptions can be traced to the specifications. The vocabulary of the specification is used in the commentary for the code—for example, to state properties that particular functions depend on to work correctly.

Testing and Inspections. Throughout the development, the deliverables of the various stages can be tested or inspected. Before executable code is available, the specifications and design documents can be formally inspected using standard industry practices.[6] The descriptions provide precise, independent criteria for correctness. In a formal inspection the reviewers know what the design or code is supposed to do and can check whether it meets these requirements.

The correctness of test cases can also be reviewed against the specifications. The behavior of the system should be predictable from the specification and tests can be chosen to verify this. Industry-standard testing techniques[7,8] can be used to supplement the tests from the specifications to catch additional code-oriented errors.

Formal Specification Languages

Rigorous software engineering requires precise but abstract descriptions of behavior. It is possible to write these descriptions in a natural language such as English, German, or Japanese, but it is very difficult to do so without introducing ambiguity and other problems. The key to rigorous software engineering is the use of formal specification languages to describe behavior.

Formal specification languages look like programming languages in that they both have special syntax and use special symbols, but they do very different jobs. A formal specification language is designed to describe *what* a product is to do while a programming language is designed to describe *how* it is to be done.

The syntax and symbols of formal specification languages are based on discrete mathematics. This can be intimidating to begin with, but in fact the math is quite simple—far easier than the continuous mathematics routinely used by hardware engineers. The math is used to provide an abstract mathematical model of the system. This is not a new idea since it is the standard approach in engineering to use a model to understand the behavior and properties of a system.

One such language is the Hewlett-Packard Specification Language, HP-SL[9]. HP-SL has been developed at HP Laboratories in Bristol, England, and has already been used on real software development projects at several divisions in HP. The language is supported by a toolset (the HP Specification Toolset), based on the `emacs` editor with an optional interface to the HP Softbench[10] product. The toolset allows the production of specification documents containing a mixture of natural language and HP-SL. Using the toolset, specifications can be checked for syntac-

tic correctness and for type correctness. The toolset is not a Hewlett-Packard product, but has been made available to academic and research institutions.

HP-SL is a specification language that uses data types, functions and logic to describe properties of software systems. These properties can be expressed at both a high level of abstraction and a high level of precision. This combination of abstraction and precision allows important behavior to be captured without becoming lost in a mass of detail.

An Overview of HP-SL

The HP Specification Language, or HP-SL, is a notation for describing the behavior of software systems and components in an abstract yet precise manner. The language allows attention to be focused on what a system does, while deferring decisions on how a system is implemented.

The following sections introduce some of the main parts of HP-SL. This should provide a useful guide to interpreting the HP-SL specifications presented in this issue.

Values

In HP-SL, values can be given names. These names can then be used to represent values. The following declaration gives the name num to the value 3.

```
val num: Int ≙ 3
```

The declaration also says that num is of type Int (integer). Int is one of the predefined HP-SL types. A declaration need not define exactly which value is chosen for a name. For example,

```
val num1 : Int
```

says that num_1 is the name for some value of type Int, but does not say which particular value is chosen. Later sections will show that logic can be used to constrain values that are given names.

Value declarations can appear at the top level of a specification (as above) in which case their scope is the entire specification. They can also be used to define local values using a let expression. The let construct is similar to that found in many functional programming languages. For example, the value of the expression:

```
let
  val one: Nat1 ≙ 1

  val ten: Nat1 ≙ 10
in
  one + ten
endlet
```

is the natural number 11. The names one and ten are defined in an inner scope and are not visible outside the let expression.

Types

Types in HP-SL are much like types in a programming language in that they are collections of similarly structured values that permit the same set of operations. Thus, numbers and Booleans (truth values TRUE and FALSE) are distinct types, since arithmetic operators (+, −, *, etc.) work on numbers but not on truth values, and the logical operators (∧, ∨, ⇒, etc.) work on truth values but not on numbers.

HP-SL provides a number of predefined types and ways of constructing brand new types from existing types. HP-SL also allows names to be given to types.

Predefined Types. Table I lists the predefined types in HP-SL. These types are available for use in all specifications.

Table I
HP-SL Predefined Types

Name	Description	Values
Bool	Boolean, truth values	TRUE, FALSE
Real	Real numbers	0, 3.142, 0.00023
Int	Integers	5, 0, 99
Nat0	Natural numbers from 0	9, 0
Nat1	Natural numbers from 1	9, 1
Char	Characters	'a', '\[newline]'
String	Sequences of characters	"abc"

The numerical types (Real, Int, Nat0, and Nat1) have all the expected arithmetical operators defined on them (+, −, *, /, <, >, mod, etc.). The Boolean type has the five standard Boolean operators defined on it (∧ logical AND, ∨ logical OR, ⇒ logical implication, ¬ logical negation, and ⇔ logical equivalence).

Naming Types. In HP-SL, types can be given additional names. The following declaration gives the name MyNum to the existing type Nat0.

```
syntype MyNum ≙ Nat0
```

This declaration does not introduce a new type, but merely gives another name to an existing type. Values of type MyNum are indistinguishable from values of type Nat0. (The keyword syntype derives from synonym type).

Constructing Types. HP-SL has some predefined type constructors that provide ways of creating more complicated types from other types. The most common of these are sets, sequences, and maps.

Sets. These types are used to model collections of data for which order and repetition are unimportant. Set types in HP-SL are constructed using the type constructor Set. For example,

```
syntype Natset ≙ Set Nat0
```

associates the name Natset with sets of natural numbers. Values of the set types are written as follows:

```
val empty_set: Natset ≙ { }
val even_set: Natset ≙ { 2, 4, 6 }
```

where empty_set is a name for the empty set, and even_set is a name for the set containing three elements 2, 4, and 6.

For large sets, it is not practical to list all the values. Therefore, HP-SL provides a syntax similar to the conventional mathematical set comprehension.

```
val one_to_a_thousand ≜ { x | x: Nat0 · x ≥ 1 ∧ x ≤ 1000 }
```

The set one_to_a_thousand contains all the natural numbers between 1 and 1000.

HP-SL provides all the conventional set operators (∪ set union, ∩ set intersection, ∈ set membership, \ set difference, and ⊆ subset).

```
3 ∈ {1...5} = TRUE
{1,2} ∪ {2,3,4} = {1,2,3,4}
{1,2} ∩ {2,3,4} = {2}
{1,2} \ {2,3,4} = {1}
{1,2} ⊆ {1,2,3} = TRUE
```

Sequences. These types are used to model collections of data for which order and repetition are important. Sequence types in HP-SL are constructed using the type constructor Seq. For example,

```
syntype Natseq ≜ Seq Nat0
```

associates the name Natseq with sequences of natural numbers. Values of this sequence type are written as follows:

```
val empty_seq: Natseq ≜ ≪ ≫
val even_seq: Natseq ≜ ≪ 2, 4 ≫
```

where empty_seq is defined to be the empty sequence, and even_seq is defined to be the sequence of two elements, the first of which is 2 and the second of which is 4.

There are a few standard names for sequence operators. The names for the HP-SL sequence operators are listed in Table II.

Table II
Operators for Sequences

Operator	Meaning	Example
hd	Head of a sequence	hd ≪ 2, 99 ≫ = 2
tl	Tail of a sequence	tl ≪ 2, 99 ≫ = ≪ 99 ≫
⊕	Sequence concatenation	≪2, 99≫ ⊕ ≪2, 99≫ = ≪2, 99, 2, 99≫
elem	Sequence lookup	(elem ≪2, 99≫ 2) = 99
len	Length of a sequence	len ≪2, 99≫ = 2

Maps. These types are used to associate values of some type with values of another type. In programming languages, maps are implemented by structures such as hash tables, association lists, and trees. In HP-SL map types are constructed directly using the type constructor $\xrightarrow{m}$. For example, the definition:

```
syntype Char_to_num ≜ Char →m Nat0
```

gives the name Char_to_num to the type mapping values of type Char to values of type Nat0. Values of this map type are written as follows:

```
val empty: Char_to_num ≜ [ ]
val amap: Char_to_num ≜ [ 'a' ↦ 2, 'm' ↦ 1, 'p' ↦ 1 ]
```

where empty is the empty map and amap is the map that associates 'a' with 2, 'm' with 1, and 'p' with 1.

The function lookup is provided to return the associated value in a map. For example,

```
lookup amap 'a' = 2.
```

Another function, dom, is provided to calculate the elements in a map's domain. For example, dom amap = {'a', 'm', 'p'}.

Introducing New Types. We have already seen ways of giving names to types using the syntype keyword. Doing this does not introduce a new type, it merely gives a name to an existing type. To introduce a new type, HP-SL provides the keyword type. This can be used in a variety of ways to construct new types, most of which are advanced topics in HP-SL. The most common use of type is to make *record* and *union* types. Record types contain values of several other types, and union types permit choices between alternatives.

For example, consider a record type used to represent boats. To represent a boat in this simple example, we only need to know the displacement of the boat and the number of engines it has. The type Boat will be adequate for this (assume we already have a type Weight to record the displacement).

```
type Boat ≜
[[ boat ▷
  (engines: Nat1,
   displacement: Weight)
]]
```

This definition states that values of type Boat have two fields: engines and displacement. The type of the engines field is Nat1, while the type of the displacement field is Weight. We can construct values of Boat using the name before the ▷ symbol—boat.

```
val single: Boat ≜ boat(1, 5000)
val double: Boat ≜ boat(2, 7000)
```

The value single is intended to represent a single-engine boat with displacement 5000, and double a twin-engine boat with displacement 7000. Just as in a programming language, the fields of a record value can be accessed. In HP-SL, the fields are accessed by functions generated from the field names used in the type definition.

```
engines(double) = 2
displacement(single) = 5000
```

Consider now a type Ship, values of which are either boats (as before) or yachts. Yachts have sails, not engines, and an important statistic is the sail area. We define this using the following union type:

```
type Ship ≜
        [[ boat ▷
        (engines: Nat1,
        displacement: Weight)
        ]]
  |
```

```
⟦ yacht ▷
(sail_area: Area,
displacement: Weight)
⟧
```

The symbol "|" is used to indicate alternatives in union types.

We can redefine the previous boats as ships:

```
val single: Ship ≜ boat(1, 5000)
val double: Ship ≜ boat(2, 7000)
val yacht1: Ship ≜ yacht(400, 2000)
```

The field accessing mechanism works as before:

```
displacement(double) = 7000
displacement(yacht1) = 2000
```

Functions

Functions are used to model computations and calculations. Functions in HP-SL are "pure" in the sense that they cannot have side effects.

Functions are defined in one of two ways—explicitly or implicitly. An explicit function definition gives a formula to calculate a result from the inputs. An implicit function definition gives a test or definition of which result is correct for the given inputs. To make this clearer, consider a function max that returns the maximum of two integers.

Defined explicitly, a function looks like:

```
1: fn max : Int × Int → Int
2: is max(x, y)
3: ≜
4: if x < y
5: then y
6: else x
7: endif
```

Line 1 gives the signature* of the function. In this example the signature says that max takes two values of type Int and returns one value of type Int. Line 2 gives names for the formal parameters of max, x and y. Line 3 introduces the body of the function definition, and says that what follows will be an algorithm to calculate the result given the inputs. Lines 4 to 7 are the explicit algorithm.

Defined implicitly, a function looks like:

```
1: fn max : Int × Int → Int
2: is max(x, y)
3: return larger
4: post
5:    (larger = x ∨ larger = y)
6:    ∧
7:    larger ≥x
8:    ∧
9:    larger ≥ y
```

Lines 1 and 2 of this definition are identical to the explicit definition. Line 3 introduces the name larger for the result. Line 4 introduces the body of the implicit definition (known as a postcondition, hence the keyword post).

*A signature defines the argument types and the result type of a function.

Lines 5 to 9 are a test which is true precisely when the value larger is the correct result for the function.

The implicit style can allow quite complicated functions to be defined in a simple way, without having to give an algorithm. For example, a square root function could be specified implicitly as follows:

```
fn sqrt: Real → Real
is sqrt(r)
return s
post
   r = s * s
```

The square root example as given is, of course, incorrect. The square root of a negative real number is not itself a real number. A specifier has two choices here—extend the definition to complex numbers or restrict the definition to nonnegative reals. A particular strength of HP-SL is its ability to restrict the scope of function definitions in a natural way. This is done by adding a precondition to the function definition. This precondition is a Boolean test that determines valid inputs to the function. In the square root example, the definition given is only valid for inputs greater than or equal to zero. Hence the corrected definition is:

```
fn sqrt: Real → Real
is sqrt(r)
pre r ≥ 0
return s
post
   r = s * s
```

Relations

For modeling operations on the system state, it is useful to identify the particular kinds of functions that return a Boolean result. Such functions are called *relations* and have a special syntax in HP-SL. For example:

```
reln vowel : Char
is vowel(c)
≜
   c ∈ {'a', 'e', 'i', 'o', 'u'}
```

defines a relation called vowel which is true just for those characters that are vowels. So the expression vowel('a') is TRUE, whereas the expression vowel('k') is FALSE. The mail system described in the article on page 32 shows how relations are used to model system operations.

Using Logic

The use of logic is pervasive in HP-SL. Logic can be used to constrain values, types, and functions to specify intended properties precisely without needing to give algorithms. The implicit form of the function definition is one way in which logic can be used.

Logic can be used to constrain value definitions by using the sat keyword. The sat is shorthand for satisfies.

```
val x: Int sat x > 10
val y: Int sat y ∈ {1,9,31}
```

This defines x to be some integer greater than 10, and y to be one of the values 1, 9, or 31.

Logic can also be used to constrain types. Consider a set of characters. To model part of a system, it may be true that these sets can never be empty. HP-SL gives a simple way of stating such constraints using the inv keyword (inv is short for data type invariant).

$$\text{syntype N\_e\_char\_set} \triangleq \text{Set Char inv s} \cdot s \neq \{\}$$

The s between the inv and the · is a name that represents a typical member of the type N_e_char_set. This name is used in the logical expression after the · to state that all such members would be nonempty. So the set {'a', '6'} would be of type N_e_char_set but the set { } would not. The notion of invariant allows powerful statements about expected properties of systems. This aids the production and analysis of specifications.

In addition to the simple logic expressions used so far (propositional logic), HP-SL also provides predicate logic. Predicate logic extends propositional logic by introducing quantifiers that allow statements about all values of some type or about some values. The symbol ∀, read as "for all" or "for any," is used to make statements about all members of some type. The expression

$$(\forall x: \text{Nat0} \cdot x \geq 0)$$

says that: for all x of type Nat0, $x \geq 0$. Another quantifier is the symbol ∃, read as "there exists," which allows statements about some values. For example, the expression

$$(\exists x: \text{Char} \cdot x = \text{'a'})$$

says that: there exists an x of type Char such that x = 'a'.

Who Has Used Rigorous Techniques?

Rigorous methods are creating increasing interest in the software development community. The articles on pages 46 and 51 describe experiences using HP-SL on real projects at two HP divisions. HP Laboratories Bristol is also providing consulting services to HP divisions in Colorado, California, and Scotland.

Organizations outside of HP have used formal specifications in various ways. For instance, one organization used formal specifications to develop a reusable framework for a family of software products for instruments.[11] They concluded that the application of formally based techniques in this fashion proved to be cost-effective, and the reusability of the specifications has translated into reusable components. Another organization used formal techniques as part of their reengineering efforts on a major software product.[12] They used the formal notation Z, which is very similar to HP–SL, to manage the introduction of new features into the product. The specification document became a record of the commitment promised by the designer to the customer. Many software houses in Europe have successfully used formal and rigorous techniques for a wide range of applications ranging from safety-critical applications like air-traffic control to telecommunications controllers.

Introducing Rigorous Techniques into a Project

For many people, rigorous techniques represent a radically different way of approaching software development. Adopting this approach can be far from straightforward. In recognition of this problem, the software engineering department at HP Labs in Bristol has a project team called the applied methods group, or AMG, whose mission is to act in collaboration with other parts of HP to introduce formally based methods.

To date, a phased approach has been taken which starts with a small, low-risk project and develops to a more substantial collaborative project. It is important that the projects have the following features:

- Enthusiasm from the project engineers to try something new
- Full effective commitment from all the managers involved, from project manager to lab manager
- Commitment to project-centered training just before the collaborative project
- Realistic expectations of what benefits and costs are involved.

In addition, the product being developed needs to have a high chance of successful delivery to market. The collaborations aim to be at the center of the product development, not merely providing an ancillary technology.

As with introducing any new technology, there are requirements on the rigorous techniques. These requirements include training, effective technical support, adequate documentation, and high-quality supporting tools that can be integrated into the normal development environment of the project. A large part of the team's work has been to ensure that this level of support is available.

Conclusion

Rigorous software engineering is an approach that is both practical and theoretically sound. Its introduction into the software engineering community is progressing. It offers scope for substantial improvements in software quality and productivity. In addition, the approach is likely to be developed further to support a wider range of applications.

References

1. D. Reinertsen, "Whodunit? The Search for the New-product Killers," *Electronic Business*, July 1983, pp. 63-66.
2. W. T. Ward, "Calculating the Real Cost of Software Defects," *Hewlett-Packard Journal*, Vol. 43, no. 4, October 1991, pp. 55-58.
3. B. Cohen, W. T. Harwood, and M. I. Jackson, *The Specification of Complex Systems*, Addison-Wesley, 1986.
4. M. Page-Jones, *The Practical Guide to Structured Design*, Second Edition, Yourdon Press, 1988.
5. M. Jackson, *System Development*, Prentice-Hall, 1983.
6. M. E. Fagan, Advances in Software Inspections, *IEEE Transactions on Software Engineering*, Vol. SE-12, no.7, July 1986, pp. 744-751.
7. G. J. Myers, *The Art of Software Testing*, John Wiley & Sons, 1979.
8. M. Ould and C. Unwin, *Testing in Software Development*, BCS Monographs in Informatics, Cambridge University Press, 1986.
9. S.P. Bear, "An Overview of HP-SL," *VDM 91 Formal Software Development Methods*, Springer-Verlay, 1991, pp. 571–587.
10. M. R. Cagan, "The HP SoftBench Environment: An Architecture for a New Generation of Software Tools," *Hewlett-Packard Journal*, Vol. 41, no. 3, June 1990, pp. 36-47.
11. D. Garlan and N. Delisle, "Formal Specifications as Reusable Frameworks," *VDM 90: VDM and Z - Formal Methods in Software Development*, Springer-Verlag, 1990.
12. C.J. Nix and B.P. Collins, "The Use of Software Engineering, Including the Z Notation, in the Development of CICS," *Journal of Quality Assurance*, Vol. 14, no. 3, September 1988, pp. 103-110.

Bibliography

1. S.L. Gerhart, "Applications of Formal Methods: Developing Virtuoso Software," *IEEE Software*, September 1990, Vol. 7, no. 5, pp. 6-10.
2. A. Hall, "Seven Myths of Formal Methods," *IEEE Software*, September 1990, Vol. 7, no. 5, pp. 11-20.
3. I. Hayes, *Specification Case Studies*, Prentice Hall International, 1990.
4. C.B. Jones, *Systematic Software Development Using VDM*, Prentice Hall International, 1986.
5. C.B. Jones and R.C. Shaw, *Case Studies in Systematic Software Development*, Prentice Hall International, 1990.
6. B. Meyer, "On Formalism in Specification," *IEEE Software*, January 1985, Vol. 2, no. 1, pp. 6-26.
7. J.M. Wing, "A Specifier's Introduction to Formal Methods," *IEEE Computer*, Vol. 23, no. 9, September 1990, pp. 8–24.

Specifying an Electronic Mail System with HP-SL

Starting with a list of system features and capabilities, an HP-SL specification for a simple mail system is developed and the steps involved in this process are analyzed.

by Patrick C. Goldsack and Tony W. Rush

Specifications tend to be used for three main purposes. The first is to help analyze the requirements of a system by constructing an abstract model. The process of constructing the model, and the subsequent reasoning about its behavior, will typically result in extensive discussion about the fundamental required behavior of the system. The second purpose is to provide concrete, unambiguous descriptions of the system that are open to detailed review. The third purpose is to act as a guide to the developers of the system by describing the necessary properties of their programs.

This paper provides an introduction to using HP-SL notation for the specification of a simple mail system. Although the mail system and its specification are simplified, enough of the system is specified to demonstrate the essential aspects of the HP-SL notation and the specification process. Most of the HP-SL notation used in this paper is described in the article on page 24.

The System

The mail system defined in this paper is similar to the electronic mail systems in common use such as HP DeskManager and the various mail systems available on systems like the HP-UX operating system. Our example mail system has the following features and capabilities.

- The system has a collection of users who are registered with the system.
- Each user has three trays: an in_tray, an out_tray, and a pending_tray.
- Users compose messages by entering them into their pending_tray.
- Users may read and delete messages from their in_tray.
- Users post messages by moving them from their pending_tray to their out_tray.
- The system transmits messages from a given out_tray to the recipients' in_tray.

A real mail system would provide many more features. However, this choice of features is sufficient to illustrate the use of HP-SL and demonstrate some of the advantages of formal specification.

Building the Specification

From the natural language description above, we can identify the following entities (data types) that must be modeled:

- Users
- Messages
- Trays (in_tray, out_tray, and pending_tray).

There are also constraints between the entities—for example, each user has three trays. We can also identify items of vocabulary that must be defined such as the meaning of each of the three tray types and the notion of being registered. Finally, several operations are identified:

- Reading a message
- Deleting a message
- Entering a message
- Posting a message
- Transmitting messages.

Message Data Types

Entities within a specification are represented as values of a type—a concept present in many high-level programming languages. However, HP-SL has a variety of types that are particularly suited to modeling requirements and behavior.

Person. The first type we define is that of person, which will be used to represent potential users.

```
type Person
```

This is an example of an abstract type. In this example, by making Person an abstract type and providing no functions that operate on values of the type, we are stating that the particular properties and attributes of the type are irrelevant to the specification. In fact, the only property we can assert about values of type Person is equality (or inequality).

Contents. The next type is that of the message contents. This is also defined as an abstract type. Later, this could be refined, for example, into a sequence of bytes. Because its structure is unimportant to the specification, the type Contents is left abstract.

```
type Contents
```

Message. To define a message, we note that there are three interesting properties of messages in the system:

- The identity of the sender of the message
- The identity of the message's intended recipients
- The contents of the message.

In addition, we need to be able to construct messages. We could define Message as an abstract type as before, and define functions to construct messages and to extract information.

```
type Message
fn message: Person × Set Person × Contents → Message
fn sender: Message → Person
fn recipients: Message → Set Person
fn contents: Message → Contents
```

The function message will construct messages and the other three functions will extract the appropriate information from values of type Message.

We could also use the following shorthand to define what these functions do.

```
type Message ≜
⟦ message ▷
      (sender: Person,
       recipients: Set Person,
       contents: Contents)
⟧
```

This type is essentially equivalent to a programming language record type. Messages are constructed using the function message and the functions sender, recipients, and contents are equivalent to field selectors in a programming language.

Validating the Message Definition

The definition of the type Message makes certain decisions regarding the behavior of the mail system which should be validated against the intended properties of the system. In any completed specification document, all formal descriptions should be accompanied by a discussion of the real-life interpretation of the mathematical model. If one looks at the information captured in each message, the following questions and considerations arise.

Sender. A single person or multiple authors? We decide here that the sender field is used to represent the name of the person who actually enters and posts the message. This can be compared to the differing approaches of HP Desk and HP-UX mail.

Recipients. The recipients are expressed as a set of type Person. Because a set has no order and no duplicates, the consequences of this choice must be explained. The lack of ordering implies that no precedence can be given to individuals within the set by virtue of their membership of the set. This seems a reasonable decision for this specification.

The lack of duplicates can be interpreted in one of two ways. First, it can mean that a message may not be created when a person occurs multiple times in the recipients field of a message. This seems unreasonable if aliases for sets of people are introduced and such aliases are allowed to overlap. Second, it can mean that duplication of a person in the recipients field has no effect on the behavior of the mail system. It is just as though that person were mentioned just once, which in this case is the intended interpretation. If one really wanted the first interpretation there are alternative ways of modeling the type that would be more natural.

Contents. This is a type for which we have no further details about its internal makeup because it has no effect on the mail system behavior. This is perhaps one of the most significant differences between this example specification and a real mail system. In practice, an arbitrary set of conventions is used to give significance to certain forms of message contents such as a line starting with the character "." or with the string "To:".

This choice of properties is the minimum necessary for this specification example. In a real system, messages would have many more properties defined, such as date sent and priority. In this specification, these would be added as additional fields of the type Message.

Defining the types used in a system builds up a vocabulary that can be used to describe the system's properties. This vocabulary can be extended to describe all aspects of the system in a completely formal and unambiguous way. This is done by defining functions that manipulate the data at a higher level. Once this extended vocabulary is in place, statements of system behavior become straightforward. This leads to one of the main benefits of formal specification—the right language with the right abstractions make discussing design issues much easier.

For messages, a useful concept is the set of people referenced within the header of a message—the sender and the set of recipients. The function addressed obtains this information from a message.

```
fn addressed: Message → Set Person is
addressed(m) ≜ {sender(m)} ∪ recipients(m)
```

The value sender(m) is a set containing only the person who sent the message. This is added into the set of recipients (using set union ∪) to produce the set of all persons mentioned in the message.

Tray Definition

The initial description of the system mentions three sorts of trays that belong to each user. The specification groups these three together as a type Trays.

```
type Trays ≜
⟦ trays ▷
    (in_tray: Tray,
     out_tray: Tray,
     pending_tray: Tray)
⟧
```

Each of the three trays is of type Tray, which is defined to be a set of messages.

```
syntype Tray ≜ Set Message
```

Note that this type declaration is slightly different from the previous ones. For a start it is introduced by the keyword syntype which stands for synonym type. A syntype introduces a name for a type, not a new type (unlike a type declaration). Thus the type expression Set Message may be used interchangeably with the type expression Tray.

Validating the Tray Definition

The decision to model the tray as a set of messages deserves closer inspection. As previously discussed, using

sets has two consequences. First, there is no order in the messages reflected within the tray's structure, and second, duplicate messages are not possible within a tray. The question to be answered is whether either of these two restrictions will cause difficulties in modeling the behavior.

Ordering the messages in a tray might reflect one of two kinds of attributes: message attributes and dynamic attributes.

Message Attributes. These are attributes that are only dependent on already existing fields of the mail messages. Examples of message attribute orderings include:

- The alphabetical order of the senders
- A priority order with the priority of a message being captured within the as yet undefined contents field of the message.

This kind of ordering can be recreated whenever it is required, so not capturing this information in the Tray type is relatively unimportant. However, in practice it would be useful to use an ordered data type if the attribute ordering were central to the operation of the system.

Dynamic Attributes. These are attributes that are the result of some action or ordering of actions. An example of this might be the order in which messages are placed in a tray. It is always possible to capture this order by having information that represents a message's position in the sequence added as an attribute of the type Message. It would be better, however, to take a much more direct approach and use, say, a sequence instead of a set.

Thus a particular choice of modeling is achieved by considering both the mathematical properties of the model and the directness with which it captures the intended interpretation of the real system. In this instance, mail messages in a tray are identified directly rather than by position in an ordered sequence, so an unordered collection seems to be a reasonable decision.

The restriction on duplicate messages is somewhat less clear. Consider the in_tray. The intention is to ensure that recipients only get a single copy of a message. However, this is already indicated for the recipients' being a set of people, and therefore, messages are only ever transmitted once to an individual. The only other way in which a message can be sent twice is for the sender to do so explicitly as two separate send operations. The question is whether we wish repeated transmissions to collapse identical messages at the receiving end. It is not clear that this is an intended aspect of the behavior of the mail system. The type used for the in_tray field could be weakened to an unordered collection with duplicates (commonly called a multiset or bag).

Note that there is a temptation at this point to say that the undefined contents field might contain a unique reference for every message placed in the in_tray, so that duplicates caused by repeated transmission in fact result in different messages. However, if this were the case, it would be an essential property of the system's behavior at the current level of abstraction and so should be modeled directly. Similar arguments for allowing duplicates may be made for the two other trays. For the purposes of this example specification, the type Tray will remain as a set to avoid introducing large amounts of HP-SL text.

Although this example may be elementary, it is typical when using formal specification languages that many questions arise both from the process of constructing the specification and from formal reviews of the specification. A style is rapidly acquired that questions every modeling decision in great detail and considers the consequences. This is only possible if the underlying modeling technique has a formal basis allowing sound reasoning.

With many data types it is convenient to define additional properties that make other parts of the specification simpler—in this case, the mail in a tray for a particular user. The function mail_for formally defines this property.

```
fn mail_for: Person × Tray → Set Message is
  mail_for (u, tray)
    return msgs
    post
      (∀ m:Message ·
        m ∈ msgs ⇔ m ∈ tray ∧ u ∈ recipients(m)
      )
```

This function returns the set of messages (msgs) in which each message m both is in the tray (m ∈ tray) and has the addressed person u as one of the recipients (u ∈ recipients(m)).

System Definition

The mail system consists of a collection of people, each with three trays. This association is specified by using a map from values of type Person to values of type Trays. The following definition states that each user can only be associated with a single collection of Trays.

$$\text{syntype Mail\_system} \triangleq \text{Person} \xrightarrow{m} \text{Trays}$$

Now that the system has been defined, the concept of a user being registered on the system can be specified. We say that a person is registered if that person is in the domain of the system mapping. Hence that person is associated with a collection of trays in the system.

```
fn registered: Person × Mail_system → Bool is
  registered (u, s) ≜ u ∈ dom(s)
```

The registered function returns a TRUE value if the person u is registered in system s. The type Mail_system will be refined later in this specification.

Operations on the System

The types of data used in the system have now been defined along with functions defining additional properties of the data. The operations listed in the informal requirements description can now be defined.

- Users compose messages by entering them into their pending_tray
- Users may read or delete messages from their in_tray
- Users post messages by moving them from their pending_tray to their out_tray
- The system transmits messages from a given out_tray to the recipients' in_tray.

Entering a Message. The relation input defines the operation of entering a message into a user's pending_tray (in preparation for sending). This will be defined as a relation between the state of the mail system before the message is entered, the message to be entered, and the state of the mail system after the message has been entered. In HP-SL, this relation looks like:

```
reln input: Mail_system × Message × Mail_system is
      input (sys, message, sys')
≜ ...
```

This relation is true if the parameters are in the relationship and false otherwise. The input parameters sys and sys' are used to indicate the before and after state of the mail system (see "Specification of State in HP-SL" on page 38).

When defining operations, it is useful to consider what constraints can be placed on the operation's parameters. It might be considered a valid restriction on the use of the input operation that the message placed in the pending_tray can only be addressed to registered mail-system users. If so, this restriction may be captured using a precondition, which is a predicate that must be satisfied by the input parameters before the operation can be used.

```
reln input : Mail_system × Message × Mail_system is
      input (sys, message, sys')
pre
   (∀ user ∈ addressed (message) · registered(user,sys))
≜ ...
```

This restriction imposed by this precondition is artificial since it does not prevent a message from being addressed to a user who might be deregistered from the system after the message is placed in the pending_tray but before transmission. A less restrictive form of the precondition would be to require that the sender of the message be registered with the system. The following definition says that the sender's pending_tray has the new message added to it and that this is the only change.

```
1: reln input : Mail_system  × Message ×  Mail_system is
2:       input (sys, message, sys')
3: pre registered (sender(message), sys)
4: ≜
5:  let
6:     val from ≜ sender (message)
7:     val trays ≜ lookup sys from
8:     val trays' ≜ lookup sys' from
9:  in
10: dom sys'  = dom sys
11:     ∧
12:    (∀ p ∈ ( dom sys \ {from}) ·
                (lookup sys' p) = (lookup sys p))
13:    ∧
14:    in_tray(trays') = in_tray (trays)
15:     ∧
16:    out_tray(trays') = out tray (trays)
17:     ∧
18:    pending_tray(trays') = pending_tray(trays) ∪ {message}
19: endlet
```

Lines 6 to 8 provide local names for:

- The sender of the message (from)
- The sender's trays before the operation (trays)
- The sender's trays after the operation (trays').

The first two clauses (lines 10 and 12) ensure that the system does not change in an unwanted way. The first clause ensures that no people are added or removed from the system (the domains of the before and after states are identical), and the second clause states that for all but the sender, the system map does not change.

The final three clauses (lines 14, 16, and 18) state how the sender's trays change (if at all), one clause for each of the three trays.

Clauses that state that the system does not change (lines 10-12) can be captured in the following relation between two states and a person.

```
reln change_only: Mail_system × Person × Mail_system is
     change_only (sys, user, sys')
≜
  dom sys' = dom sys
  ∧
  ( ∀ p ∈ (dom sys' \ {user}) · (lookup sys' p ) = (lookup sys p))
```

Given this definition, input can be rewritten more succinctly as:

```
reln input: Mail_system  × Message  × Mail_system is
     input (sys, message, sys')
pre
  registered (sender(message), sys)
≜
   let
     val from ≜ sender (message)
     val trays ≜ lookup sys from
     val trays' ≜ lookup sys' from
   in
     change_only (sys, from, sys')
     ∧
     in_tray(trays') = in_tray(trays)
     ∧
     out_tray(trays') = out_tray(trays)
     ∧
     pending_tray(trays') = pending_tray(trays) ∪ {message}
   endlet
```

Deleting a Message. The operation delete removes a message from a user's in_tray. The operation takes a user (of type Person) who is deleting the message and a message to be deleted as parameters. The new state differs from the old only in that the message has been removed from the user's in_tray. The form of the definition is very similar to the relation input.

```
reln delete: Mail_system × Person × Message × Mail_system
is
     delete(sys, user, message, sys')
pre
   registered (user,sys)
   ∧
   message ∈ in_tray (lookup sys user)
≜
   change_only (sys, user, sys')
   ∧
   in_tray (lookup sys' user) = in_tray (lookup sys user)\
        {message}
   ∧
   out_tray (lookup sys' user) = out_tray (lookup sys user)
```

```
    ∧
    pending_tray (lookup sys' user) = pending_tray (lookup sys
          user)
```

This relation has a precondition that states two things:

- The user is registered (in the domain of the system map)
- The given message is initially in the user's in_tray.

Given that the precondition is satisfied, the remainder of the relation specifies that only trays associated with the user should be changed and that the message should be deleted from the user's in_tray, but the user's out_tray and pending_tray remain unchanged.

The second part of the precondition might be considered overly restrictive. The behavior specified in the precondition is actually valid whether the message was there at the start or not (removing a nonexistent element from a set leaves the set unchanged). However, this would imply that the correct behavior is silently to do nothing when the user asks to delete a nonexistent message. In this specification we wish to leave open the possibility of some other error behavior so we do not fully define the operation.

Posting a Message. When a user wishes to move a message in the pending_tray to the out_tray for delivery, the post_message operation is used.

```
reln post_message: Mail_system × Message × Mail_system is
     post_message (sys, message, sys')
pre
   registered (sender(message),sys)
   ∧
   message ∈ pending_tray (lookup sys (sender message))
≙
   let
      val from ≙ sender (message)
   in
      change_only (sys, from, sys')
      ∧
      in_tray(lookup sys' from) = in_tray(lookup sys from)
      ∧
      out_tray(lookup sys' from) = out_tray(lookup sys from)
      ∪ {message}
      ∧
      pending_tray(lookup sys' from) =
      pending_tray(lookup sys from) \ {message}
   endlet
```

The specification states that the message is deleted from the user's pending_tray and added to the user's out_tray. No other parts of the system are changed.

Transmitting a Message. The next specification defines the action of the system in transmitting all messages in a particular out_tray to their intended recipients' in_trays.

```
1: reln transmit: Mail_system × Person × Mail_system is
2:      transmit (sys, user, sys')
3: pre
4:    registered (user,sys)
5: ≙
6:    dom (sys') = dom (sys)
7:    ∧
8:    out_tray(lookup sys' user) = { }
9:    ∧
10:   (∀ p ∈ dom(sys)
11:     pending_tray(lookup sys' p) = pending_tray(lookup sys p)
12:       ∧
13:       in_tray(lookup sys' p) =
14:          in_tray(lookup sys p) ∪
15:                mail_for(p, out_tray(lookup sys user))
16:       ∧
17:       p ≠ user ⇒ (out_tray (lookup sys' p)
18:                = out_tray (lookup sys p))
19:   )
```

This specification has a slightly different format from the others. All users are treated identically with respect to their in_trays (in lines 14 and 15 the messages for the user are added to the in_tray) and their pending_trays (in line 11 the pending_trays do not change).

The behavior with respect to the out_trays differs between the sender and the other users. The sender's out_tray is left empty (line 8). The other out_trays are unchanged (line 17), care being taken to ensure that this clause does not apply to the sender by guarding it with the condition p ≠ user.

The clause on line 6 ensures that no one is added or removed from the system by this operation.

Because every user's trays are potentially altered, the change_only relation cannot be used.

Reading a Message. The final operation is that of reading messages. This operation is modeled by the function messages_of, which returns the set of messages contained in a user's in_tray.

```
fn messages_of: Mail_system × Person → Set_Message is
   messages_of (sys, user)
pre
   registered(user,sys)
≙
   in_tray(lookup sys user)
```

Reasoning About the System

The primary advantage of formal specifications over informal specification techniques is the ability to reason about the properties of the specification. This is done for two reasons:

- Since statements about behavior can be written in a formal language for which there is a sound set of inference rules, it can be determined if these statements are consistent .
- It can be determined when a specification is complete (when the model has been fully defined at the chosen level of abstraction). At this point, it should be possible to answer all questions regarding pertinent behavior.

Neither of these properties holds for informal techniques. In a normal software lifecycle it is not until the code is written (the first complete formal description) that many questions regarding behavior can be answered with any certainty, either by code reviews or by testing. This is typically too late in the project, and contributes to many of the problems of software development.

Many questions can be answered purely by inspection of the formal specification just as programs can be understood by examining code. However, formal techniques

also allow properties to be verified formally. This is accomplished by stating a particularly property required of the system and then proving that the specification satisfies this property.

Formal verification in this style is difficult and time-consuming, and current tool support for reasoning is extremely inadequate. In practice, it is not expected that users of formal techniques carry out this process in detail. However, it is always useful to understand how to consider properties formally even when carrying out informal, but rigorously based, arguments.

As an example, suppose we wish to show that all messages in each of the system's in_trays are addressed to the owner of that tray. The property can be stated in HP-SL with the clause:

```
( ∀ sys: Mail_system ·
 ( ∀ person ∈ dom(sys) ·
  ( ∀ msg ∈ (in_tray(lookup sys person)) ·
    person ∈ recipients(msg)
  )))
```

which says that for any mail system sys and for every person registered in that mail system, each person is a recipient of all the messages (msg) in that person's in_tray.

This property can be easily disproved by construction of a counterexample—say a system with one user called A whose in_tray contains a message addressed to user B.

What we actually need to show is that this is a property that is guaranteed to be maintained by the operations provided. In other words, if a mail system is in a state where no message has been misassigned to an in_tray then the operations provided will not cause this to happen. In addition, since this property is clearly true of an empty mail system (one with no messages), and mail systems are only created by application of the operations described, then we know that for all achievable states of the mail system this will be true (a proof by induction).

The proof is trivial because the only operation that adds any values to the in_tray is the transmit operation, which clearly maintains the required property. The argument for the proof runs as follows:

- The initial state of the system has the property that all users have empty trays
- A user's in_tray is modified by the addition of messages generated by mail_for applied to this user
- mail_for returns only mail that has the user in the recipients field
- Hence the property is maintained.

Proofs can be fully elaborated to the required degree of formality. However, this kind of semiformal rigorous argument is much more common.

Adding System Invariants

If a property such as that just examined is true for all achievable mail systems, it is useful to highlight it by stating that it is a system invariant (a property or relationship that must hold for all instances of a particular type). In HP-SL, this can be done by modifying the type definition.

```
syntype Mail_system ≙
  (Person ⇸ Trays)
  inv sys ·
   ( ∀ user ∈ dom(sys) ·
      ( ∀ msg ∈ in_tray(lookup sys user) ·
          user ∈ addressed( msg)))
```

The definition of Mail_system is similar to the earlier definition except that a logical constraint has been added. This constraint is specified by stating that the logical expression following the · is true for all possible states of the mail system sys. The logical expression states that the only messages in users' in_trays are messages that are addressed to them.

The type definition with the added invariant asserts that a mail system must never get into a state where the invariant is false. It is required that every operation on the mail system does not break this condition. As we have seen, this is true of all the operations so far defined. In addition, the definition of each operation can assume that the invariant holds when the operation is called.

In fact, another invariant property exists for the mail system. This is that the out_tray and pending_tray of a person can only contain messages that have that person as the sender. Hence the system type definition with the complete invariant is:

```
syntype Mail_system =
  (Person ⇸ Trays)
    inv sys ·
     ( ∀ user ∈ dom(sys) ·
       ( ∀ msg ∈ in_tray(lookup sys user)
       user ∈ recipients(msg))
        ∧
       ( ∀ msg ∈ out_tray(lookup sys user) ∪
         pending_tray(lookup sys user) · user = sender(msg))
    )
```

Analysis of the system and its intended behavior may well reveal other invariant properties, but for this example specification these two will suffice.

Reasoning is, of course, not limited to invariant properties. Other aspects of behavior can be analyzed. For example, the specification has made certain assumptions about the behavior of the system when posting a mail message that contains an unregistered recipient. The formal specification gives a way of deciding if assumptions such as these are reasonable.

Limitations of the Specification

Now that we have a have a specification of a mail system, there are several problems with this model of a mail system.

- The model is lacking many important operations. It should be clear that many more operations on the mail system could be added. However, for this paper relatively little benefit would be gained so the additional definitions have been left out for the sake of brevity.

Specification of State in HP-SL

HP-SL is primarily a functional specification language. There is no direct concept of state variables within the language. Hence there can be no notion of side effect, no concept of sequence, and no statements—only expressions. This adds to the clarity of specifications and facilitates reasoning.

This does not mean that we cannot describe state dependent systems in HP-SL. State-related behavior can be modeled in a completely straightforward way. If we analyze the use of state variables in programming languages, we see that there are just two distinct forms:

- Local variables used in algorithms to evaluate the result efficiently
- The system state, which is the persistent data that is kept and used by the system operations.

The first of these uses is not relevant when dealing with the style of specification used in HP-SL. HP-SL specifications make assertions about behavior with no concern for run-time efficiency. Efficiency is seen as a separate concern which is addressed during design, not specification.

However, system state is useful because it gives a very natural style of describing, designing, and implementing large numbers of systems. To specify system state in HP-SL, all that is necessary is a type that can represent all the information necessary to specify the operations on the system. Hence, every possible state of the system can be represented as a value of this type.

Every state-modifying system operation is defined as a relation between the initial state (the state existing before the operation), any parameters necessary for the operation, the result returned by the operation (if any), and the final state (that resulting from the call of the operation).

The typical form of such a relation is:

```
reln a_relation : System  × ... × System
is   a_relation (sys, ... , sys')
≙
...
```

By convention, sys is the name used for state and the initial value of the state is distinguished from the final value by the use of an apostrophe ('). Thus, the final value is called sys'.

- The model is inadequate because it does not deal with issues of concurrency. The specification assumes that every operation on the mail system is complete before another can start. This is clearly not a reasonable restriction. Why should users not add messages to their pending_trays while others are transmitting the messages in their out_trays.

  This concurrency could be modeled at the expense of greater complexity in the specification. If the purpose of the specification is to reason about concurrent behavior, then this should be done as a separate step.

- The model is inadequate because it does not deal with an unreliable communication medium.

  None of these limitations invalidates the specification when considered as a specification of message addressing, tray manipulation, and so forth. It does describe an abstract view of some properties of the mail system but it makes no claim about defining all behavior.

  In principle, very detailed models of systems and their properties can be defined. However, these are rarely the best specifications to write at least as an initial system specification. In particular, detailed specifications can mask overall abstract behavior and make it harder to reason about a system's properties. It is part of the skill of using notations such as HP-SL to decide which aspects of the system are important and which may safely be deferred. Specification can be used in just those areas that are considered likely to be difficult. The specification process may be repeated at several levels of detail and at different points in the project.

Refinement

An important feature of using formal specifications is the ability to write specifications of the system in increasing levels of detail and to reason about the relationship between such specifications. The process of constructing a more detailed specification from a more abstract one and demonstrating their correspondence is called refinement.

Refinement is rarely done in practice for whole systems (it takes too much time) but an abstract specification of a whole system, followed by detailed specifications of one or more key subsystems is practical and useful. As with reasoning about system properties, reasoning about the correctness of refinement steps can be carried out in an informal or semiformal manner as required.

In refinement, one of the key principles is to show that the properties of the abstract specification are maintained in the second, more concrete specification. For example, in moving to a more concrete representation of messages, one might expand the definition of a message to include other fields. This moves the abstract specification closer to a representation that can be modeled directly in an implementation. So, for example, a sequence could be used instead of a set since it is typically easier to implement a list in a programming language than a set.

```
type New message ≙
    ⟦new message ▷
        (new_sender: Person,
         new_recipients: Seq Person,
         creation_date: Date,
         sending_date: Date,
         text: Contents
        )
    ⟧
```

The correspondence between this model for a message and the one given earlier for type message is demonstrated by the use of a function (technically called a retrieve function) which shows how the new type represents the old type. This is part of a process that shows that the more complex type is capable of representing all possible values of the more abstract type.

```
fn retrieve_message: New message → Message is
  retrieve_message(new_msg)
  return msg
  post
      sender(msg) = new sender(new_msg)
      ∧
      recipients(msg) = elems(new recipients(new_msg))
      ∧
      contents(msg) = text(new_msg)
```

Given this, the process of demonstrating the correctness of this step continues with showing that the new specifi-

cation's operations behave equivalently to those of the original abstract one. Unfortunately, this is too complex to demonstrate and explain in an introductory paper.

In principle, refinement could continue all the way to code, thus ensuring that the final code implements the original specification. Unfortunately the technology needed to help with the proofs is not yet available and so this is not feasible for whole complex systems. What refinement does give is a definition of what it means to implement abstract data types and operations. This definition can be informally checked as part of the process of software construction.

Conclusion

This paper has shown how HP-SL formal notations can be used in the specification process by discussing a simple example. In particular:

- Creating such specifications forces the specifier to confront questions regarding the behavior of the system that might otherwise be overlooked.
- The ability to reason about the specification is important to provide a firm base for specification reviews and for subsequent coding.
- The clarification of understanding of a system forced by the search for appropriate abstractions is an essential part of the system definition.
- The notation provides a clear and unambiguous statement of behavior, without the confusion of discussing mechanisms and algorithms.

Specifying Real-Time Behavior in HP-SL

Using the event and history specification features of HP-SL, the software for a real-time alarm monitor is specified.

by Paul D. Harry and Tony W. Rush

Many of the systems that HP builds must be able to exhibit real-time properties such as concurrency. Therefore, it is important to be able to specify not just what happens in a system, but also when events happen. This paper provides an example of using a feature of the HP Specification Language (HP-SL) called history types to specify an alarm monitor for an electrocardiogram (ECG).

Alarm Monitor

As part of the efforts to work with divisions in HP's Medical Products Group, we in the applied methods group of HP's Bristol laboratory were presented with the following specification for a raise alarm system monitor (alarm monitor) for ECG measurements.

- The system monitors incoming ECG measurements and compares them with maximum and minimum limits.
- The alarm is initially canceled.
- If the incoming measurement exceeds the alarm limits for more than a predefined timeout (delay) period, the alarm should be set.
- When an in-limits value is received the alarm is canceled.
- The user can enter new alarm limits at any time.
- There are initial (default) limits.

From this informal specification we can identify three states the raise alarm system can be in (normal, delay, and alarm), and two values of the alarm (alarm_set and alarm_canceled). A constant, t_delay, is the time for which all the measurements have to be out of limits before an alarm is raised. Fig. 1 shows the relationship between measurements, alarm limits, states, and the alarm values.

Alarms, Measurements, and Limits

From the natural language specification, the following HP-SL data type can be defined:

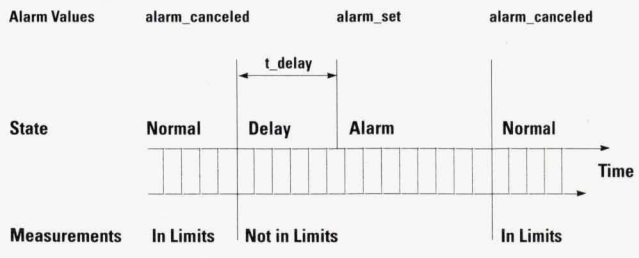


Fig. 1. Measurements, states, and alarm values.

```
syntype Alarm_limits ≙ Real × Real inv (min, max) · min < max
```

This type consists of pairs of real numbers, with the added constraint that the first member of the pair must be strictly less than the second. This is a use of the tuple data type with an invariant* subtype. Therefore, the value (9, 100.5) is of type Alarm_limits but the value (9, 2.5) is not.

There is some default value for the alarm limits. We shall merely state that this constant is of type Alarm_limits, but not give an actual value. This is an example of underspecification.

```
val default_limits: Alarm_limits
```

The alarm itself can be either set or canceled.

```
type Alarm ≙ ⟦ alarm_set ⟧ | ⟦ alarm_canceled ⟧
```

The type Alarm is modeled as an enumerated type with the two (distinct) values alarm_set and alarm_canceled.

The incoming measurements, which are constructed from the ECG wave, are simply real numbers.

```
syntype Measurement ≙ Real
```

A given measurement is either within the alarm limits or not. This can be observed by the function in_limits.

```
fn in_limits : Measurement × Alarm_limits → Bool
is
in_limits(value, (min, max))
≙
min ≤ value ∧ max ≥ value
```

So, for example, the expression: in_limits(5, (2, 9.5)) is true, whereas the expression: in_limits(1, (2, 9.5)) is false.

The raise_alarm Process

We can specify this system by the raise_alarm process. Fig. 2 is a first attempt at drawing a data flow diagram for this process. The process takes two inputs (the alarm limits and the measurement) and produces one output (the alarm state). In the data flow diagrams of structured analysis, the flows consist of individual values. For example, the meas flow consists of individual values of type Measurement.

The behavior of the raise_alarm process depends not just on one measurement, but rather on many previous mea-

*An invariant is a property that must always be maintained for a particular type.

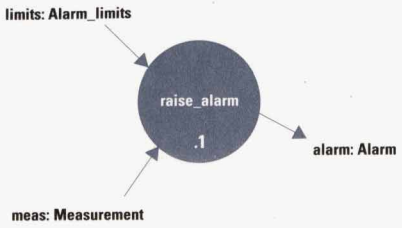


Fig. 2. Initial data flow diagram of the raise_alarm process.

surements. In other words, the behavior depends on the entire history of measurement values. In structured analysis the necessary historical information would have to be saved within process raise_alarm, which would be done by decomposing the process into subprocesses and a data store. Fortunately, there is a more direct way of specifying the behavior using a style of specification known as history specification. Instead of defining a process in terms of the current values on its data flows, history specifications define the process in terms of all the past values on the flows, that is, in terms of histories of values.

History Types

There are two forms of histories, event histories and state histories, each reflecting different time properties.

Event Histories. Event histories model flows that have events occurring at discrete times. Examples of event histories include:

- Pulses from a revolution counter on a car drive shaft
- Debits and credits on a bank account
- Button presses
- The incoming measurements in the ECG alarm system.

Event histories are characterized by the significance of duplication. For example, two debits to a bank account (even if they have the same value) are different from one debit. Event histories can be illustrated by a timeline as shown in Fig. 3.

The timeline diagram shows:

- The start time and end time of the history. In Fig. 3 the start time is t0, and the end time is t5. This gives the time interval over which this history models the events.
- The events, shown as X symbols. Each event has a value (shown above the line) and a time (shown below the line). For example, in Fig. 3 there is an event with value b at time t3.

Event histories are modeled by the HP-SL type constructor Hist_e. For example, a history of measurement events would have type Hist_e(Measurement).

The expression times(h) is the set consisting of all the times of events in history h. Using the history h_e defined in Fig. 3:

$$\text{times}(h_e) = \{ t2, t3, t4 \}$$

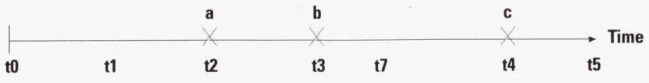


Fig. 3. An event history, h_e.

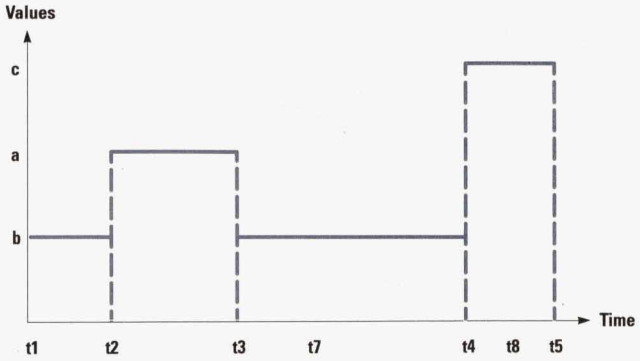


Fig. 4. A state history, h_s.

The operators start_e and end_e return the start and end times of an event history. From the example history in Fig. 3,

$$\text{start}_e(h_e) = t0$$
$$\text{end}_e(h_e) = t5$$

Given an event history h, we can find the value of the most recent event relative to some time t using the expression previous_event(h, t). The result is undefined if there is no event at or before time t. Returning to the example history:

$$\text{previous\_event}(h_e, t3) = b$$
$$\text{previous\_event}(h_e, t7) = b$$
$$\text{previous\_event}(h_e, t1) \text{ /* not defined */}$$

Finally, a time range can be checked using the operator in_interval_e. For example, the expression

$$t \text{ `in\_interval}_e \ h$$

tests whether a time t is within the range of times covered by the history h. From Fig. 3,

$$t7 \text{ `in\_interval } h_e = \text{TRUE}$$
$$t6 \text{ `in\_interval } h_e = \text{FALSE}.$$

State Histories. State histories model flows that always have a current value. Examples of state histories include:

- Distance traveled by a car
- The balance in a bank account
- The current condition of a switch (e.g., on or off)
- The alarm limits in the ECG alarm system
- The state of the alarm in the ECG alarm system.

State histories are characterized by the insignificance of duplication. For example, if the bank balance is overwritten twice in succession with the same value, this is indistinguishable from overwriting once. Further, state histories always have a possibly underspecified initial value. State histories can be illustrated by the graph shown in Fig. 4.

State histories are modeled by the HP-SL type constructor Hist_s and just like event histories, state histories have endpoint operators written as: start_s and end_s. The current value of a state history is found by the expression h @ t. Since state histories always have a current value, this operator can be applied at any time within the endpoints of the history. The following formulas hold for the history illustrated in Fig. 4.

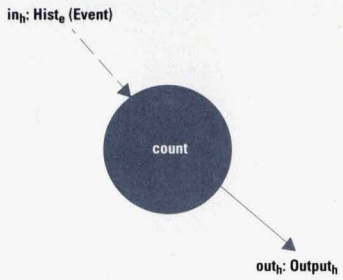


Fig. 5. A count process.

$start_s(h_s) = t1$
$end_s(h_s) = t5$
h_s @ t7 = b /* at time t7, h_s has value b */
h_s @ t4 = b
h_s @ t8 = c
(∀ t: Time sat t3 ≤ t ∧ t < t4 · h_s @ t = b)

Note that the use of sat on the last line constrains t to be between t3 and t4.

Specifying Processes

In history specifications processes are specified by relating entire histories of inputs to histories of outputs. In Fig. 5 the process count has input of type $Hist_e$(Event), and an output of type $Output_h$. The input models a stream of events. The count process outputs a continuously updated count of the number of input events received. Since the output is continuously updated, it is modeled by a state history. The following invariant ensures that the initial value of this history is zero.

syntype $Output_h$ = $Hist_s$(Nat0) inv h · h @ $start_s$(h) = 0

Using an HP-SL relation the count process is specified as follows:

reln count : $Hist_e$ (Event) × $Output_h$
is count(in_h, out_h)
≙
let
  val now ≙ $end_e(in_h)$
in
  out_h @ now = number_of_events(in_h)
endlet

The local value now is set to the end time of the history of input events. The left-hand part of the main expression, out_h @ now, is the current value of the output state history. The right-hand part, number of events(in_h), is the number of events in the input history.

This is the usual form in which we present history specifications. However, the history specification alone does not say enough. The output history is only partly defined because we have defined its value at the end time, but have said nothing about its values at earlier times. We really want the relation between the current output value and the number of events applied throughout the output history. To do this we should surround the above relation with a universal quantification (∀, read as "for all") over all times along the histories. Fortunately, the notation used in history specifications does this for free, so there is no need to write it ourselves.

The overall meaning of the count process specification is therefore:

> At any time, the output history of the system (out_h) is correct with respect to the given input history (in_h) if the current value of the output history equals the number of events in the input history.

Using the Histories

Fig. 6 shows the data flow diagram for the raise_alarm process with the correct types on the data flows. Dotted lines are used for event histories and solid lines for state histories.

The type definitions for the input and output histories shown in Fig. 6 are defined as:

syntype $Measurement_h$ ≙ $Hist_e$(Measurement)
syntype $Limits_h$ ≙ $Hist_s$(Alarm_limits) inv h ·
  h @ $start_s$(h) = default limits
syntype $Alarm_h$ ≙ $Hist_s$(Alarm)

Note that the types $Limits_h$ and $Alarm_h$ are declared to be state histories, whereas the type $Measurement_h$ is declared to be an event history. This is a suitable modeling choice because there are initial values for the alarm and both alarm limits, whereas there is no initial value for measurements. Indeed we shall see that one of the functions, in_limits_h, needs to test whether there have been any measurements before a given time. Such a test is not meaningful for a state history.

We can now start to define the process by the relation:

reln raise_alarm : ($Measurement_h$ × $Limits_h$) × $Alarm_h$
is raise_alarm((meas, limits), alarm)
≙ ...

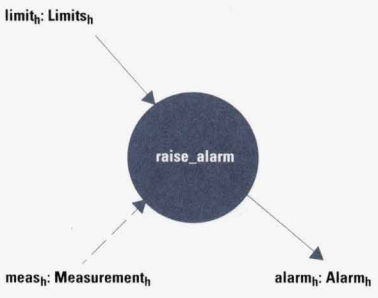


Fig. 6. Data flow diagram of the raise_alarm process with the correct data flows.

History Specifications

The ECG alarm example shows only a small part of the power of history specifications. Other features include:

Unified Graphical Notation. History specifications combine the accessibility of a graphical notation with the power of a textual one. Special graphical notations are provided for the most commonly used process specification idioms.

Composition. Processes can be composed together to specify larger systems. For example, we could add processes to the alarm system to change the alarm limits, and to veto the alarm. This feature is illustrated by the composition diagram shown in Fig. 1.

Specification Styles. The state and operations style of specification used for the mail system (see page 32) can be incorporated into history specifications. For example, the composition diagram for the mail system is shown in Fig. 2.

The following guidelines could be used to determine which specification style to use.

- Use state-based specifications when:
  - The system is similar to a database. It is surprising how many systems can be partitioned into a central data store accessed by read and update operations.
  - The system has few properties dependent on time.
  - The structure of the system is sufficiently simple not to need a graphical index to the system.

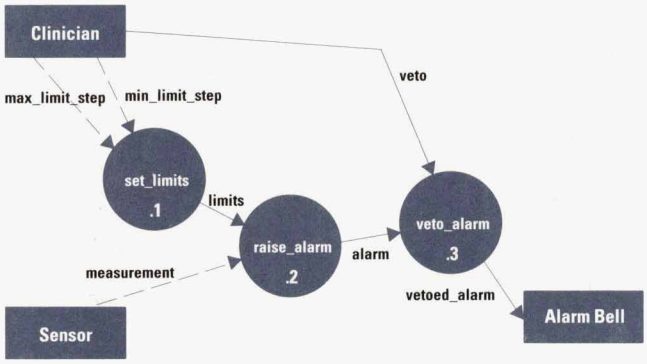


Fig. 1. Composition diagram for the alarm system.

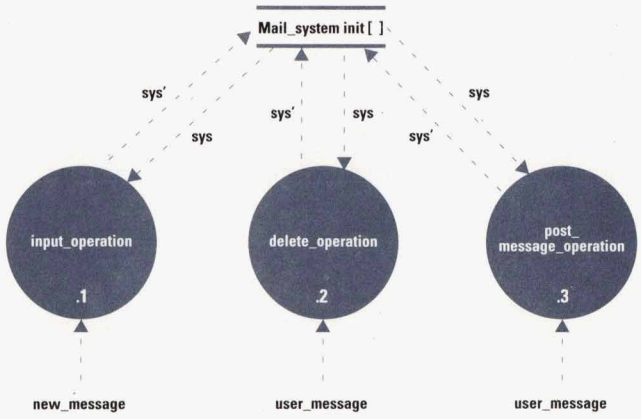


Fig. 2. Composition diagram for the mail system.

- Use history specifications when:
  - The system has many properties dependent on time.
  - The hierarchical structuring of data flow diagrams is adequate, but their ambiguity and limitations are not.
  - The system is structured into a core and concentric outer fringes.

Implementation route. There is a systematic implementation route from specification to code via structured design. This route consists of five steps:

- The starting point is the history specification in which processes relate whole histories of values.
- The history specification is treated as a classical data flow diagram and transformed into a structure chart. This structure chart will also be in terms of whole histories.
- No implementation can handle histories, so each history must be replaced by the data that needs to be stored. For example, in the count process the history of incoming events is replaced by a count so far. The result is known as the reduced design.
- The reduced design is optimized.
- Module specifications are written. This is usually only necessary when optimization has made it hard to understand the modules from the original process specifications.

Detecting Out of Limits Values

In an earlier section that described alarms, measurements, and limits, a Boolean function, in_limits, was defined which is true when a given value of the ECG measurement is between given alarm limits. The alarm monitor's behavior depends not just on whether the current measurements are in limits, but also on whether older measurements are in limits. To do this we define a function in_limits_h over the histories Measurement_h and Limits_h. This function returns TRUE if, at a given time t, the measurement is in limits, or if there have been no measurements.

$$\text{fn in\_limits}_h : \text{Measurement}_h \times \text{Limits}_h \times \text{Time} \rightarrow \text{Bool}$$
$$\text{is in limits}_h(\text{meas, limits, t})$$
$$\text{pre}$$
$$(\text{t } \text{`in\_interval}_e \text{ meas}) \wedge (\text{t } \text{`in\_interval}_s \text{ limits})$$
$$\triangleq$$
$$(\exists\, \text{t1} \in \text{times(meas)} \cdot \text{t1} \leq \text{t})$$
$$\Rightarrow$$
$$\text{in\_limits(previous\_event(meas, t), limits @ t)}$$

The precondition uses the functions `in\_interval}_e$ and $\text{`in_interval_s to ensure that the histories are defined at the given time, t.

Deciding if there have been any measurements yet is done by the expression:

$$(\exists\, \text{t1} \in \text{times(meas)} \cdot \text{t1} \leq \text{t})$$

which is true just when there is at least one event at some time t1 in the history meas earlier than time t. (The existential operator $\exists$ is read as "there exists" or "there exists one or more.")

The function returns a default value of TRUE if there are no measurements before the given time t. If there are earlier measurements, then the value of the current measurement is found by using the previous_event operator. The current limits are found by using the current value opera-

tor (@) at the given time t. These measurement and limit values are then compared using the in_limits function.

Alarm Detection

From the natural language specification for the alarm monitor it is clear that:

- The alarm is only set after all measurements have been outside the alarm limits for a given time
- Any occurrence of an in-limits value cancels the alarm.

The value of the time delay before out-of-limits values cause an alarm is not important to this specification. Instead, a constant is defined without specifying an associated value.

```
val t_delay: Time
```

In practice, the value of this constant would be constrained to be within clinically acceptable limits.

For an alarm to be raised at a particular time, called now, the value of the measurement history must have been consistently out of limits for a time t_delay. Hence at any particular time, a key value is the portion of the measurement history occurring between now and now – t_delay. In fact, of most importance is the value of the function in_limits_h at all times between the interval now and now – t_delay. Informally, an alarm will be raised only if in_limits_h is FALSE at all times in this time interval. So, for an alarm to be raised at time now, the following condition should be TRUE.

```
(∀ t: Time sat (now − t_delay) ≤ t ∧ t ≤ now ·
 ¬ in_limitsh( meas, limits, t ) ) /*¬ = unary operator not*/
```

This is not quite enough since it fails to take into account the behavior of the system when it starts, that is, when now – t_delay ≤ start_time. The normal way to specify such a condition in HP-SL is to use the existential quantifier ∃. Informally we can say that an alarm condition exists if (and only if) the system has been operating for at least time t_delay, and for all times between now and now – t_delay the measurement was not in limits. This can be expressed formally as follows:

```
( ∃ ts = (now − t_delay) sat ts > start_time ·
   ( ∀ t: Time sat ts ≤ t ∧ t ≤ now ·
    ¬ in_limitsh( meas, limits, t )))
```

This expression is true only when we can construct a time (t_s) that is t_delay before now and after the start time, such that the value of the measurement history is out of limits at all times between t_s and now. In particular, the expression cannot be true until now is greater than t_delay (because no valid t_s can be constructed).

Final Specification

The following is the final specification for the raise_alarm process.

- Alarms, limits, and measurements:

```
syntype Alarm_limits ≙ Real × Real inv (min, max) · min < max
val default_limits: Alarm_limits

type Alarm ≙ ⟦ alarm_set ⟧ | ⟦ alarm_canceled ⟧
syntype Measurement ≙ Real /*ECG measurements from
                             patient*/

fn in_limits : Measurement × Alarm_limits → Bool
is in_limits(value, (min, max))
≙
min ≤ value ∧ max ≥ value
```

- History definitions:

```
syntype Measurementh ≙ Histe(Measurement)
syntype Limitsh ≙ Hists(Alarm_limits) inv h ·
                    h @ starts(h) = default_limits
syntype Alarmh ≙ Hists(Alarm)
```

- Detecting out-of-limit values:

```
fn in_limitsh : Measurementh × Limitsh × Time → Bool
is in limitsh(meas, limits, t)
pre
   (t 'in_intervale meas) ∧ (t 'in_intervals limits)
≙
 ( ∃ t1 ∈ times(meas) · t1 ≤ t)
⇒
 in_limits(previous_event(meas, t), limits @ t )
```

- Alarm detection:

```
val t_delay: Time /* duration of out-of-limit values needed
                     to raise the alarm*/
reln raise_alarm : (Measurementh × Limitsh) × Alarmh
is raise_alarm((meas, limits), alarm)
≙
let
  val now = ende(meas)
  val start_time = starte(meas)
in
  if
    ( ∃ ts = (now − t_delay) sat ts > start_time ·
       ( ∀ t: Time sat ts ≤ t ∧ t ≤ now ·
        ¬ in limitsh(meas, limits, t)))
  then
     alarm @ now = alarm_set
  else
     alarm @ now = alarm_canceled
     endif
endlet
```

Exploring the Specification

One of the advantages of specifying a system formally is that the specification provides an opportunity to reason about the system and the consequences of the choices made. For instance, some of the questions that can be raised about this example include:

- Is the system's behavior at startup correct?
- What exactly happens when no ECG signal is received?
- What happens when alarm limits are changed?

On startup, no alarm can be raised. This is ensured by the test in the if expression of raise_alarm being false. Hence the output value is alarm_canceled. This would seem to be correct when evaluated against the informal requirements.

When no ECG signal is received, the value of in_limits_h is found by looking backwards in the measurement history for the last received measurement. Hence, if the last measurement value was out of range, an alarm will be raised after t_delay. If the last value was in range, no alarm will be raised. If there have never been any values, the default behavior of in_limits_h ensures that no alarm is raised.

When limits are changed, the function in_limits_h automatically uses the new value of the limits to test incoming values. An interesting case is when the limits are changed and no ECG measurements are received. In this case, the most recent value on the ECG input is compared with the new limits. This has the consequence that an alarm might be raised even though no new ECG data is received because the limits have been changed. Is this the correct behavior?

The answer is that this is a feature of the product that needs to be decided. This question can be fed into the normal product requirements definition activity. The use of a more formal specification highlights these boundary cases at an early stage in the software development process.

Conclusion

This article has introduced the history data types of HP-SL, and has shown how they are used in history specifications to specify processes. We have found that in practice this style of specification works extremely well in describing embedded systems. For example, see the article "Formal Specification and Structured Design in Software Development" on page 51.

History specifications are an attempt to combine the best features of formal specifications and structured analysis. Like structured analysis, they have an accessible graphical notation that allows a large system to be decomposed into a hierarchy. Like formal specifications, they have a rich data language and a fully defined meaning, and support abstraction or underspecification. In addition, history specifications allow properties involving time to be stated very directly. A problem that might require a control specification, a data store, and several subprocesses in structured analysis may only require a couple of functions when using histories.

Using Formal Specification for Product Development

In one product development project, the use of precise software specifications helped to uncover potential problems that might ordinarily be overlooked, and raised some interesting issues about using formal techniques.

by B. Robert Ladeau and Curtis W. Freeman

Early in 1989 a collaboration was set up between a project team at the cardiac care systems (CCS) business unit at HP's Waltham Division and the applied methods group (AMG) at HP Laboratories in Bristol, England. The collaboration involved project engineers from both groups, with communication taking place through a few on-site visits and a lot of electronic-mail correspondence.

At a very high level, the goal for both groups was to improve the quality of the software that was to be developed. There were additional goals specific to each group. We at Waltham were interested in adding more discipline to our software development process and learning more about formal methods. Goals specific to the AMG were to transfer the latest software development technologies to other HP divisions. In our case this meant enough training and support in the HP Specification Language (HP-SL) to enable us to write our own formal specifications. In return, the AMG engineers would get feedback from projects within HP product divisions specific to their research interests at HP Laboratories.

This paper reviews the results of our collaboration involving the introduction and use of formal specification during the development of a medical product software enhancement. We discuss the lessons learned during this process of introducing an advanced software engineering methodology into an R&D environment. We also describe the specific achievements and problems that were experienced in using formal methods to specify parts of the software functionality.

The Project

The project at Waltham was created to add a new feature (ST segment measurement) to an existing medical product. The product is a bedside monitor that is used to monitor vital signs of patients in intensive care units and operating rooms. ST segment measurement is a measurement of the change in a portion of a patient's electrocardiogram (ECG) called the ST segment (see Fig. 1). Changes in the ST segment of a person's ECG can indicate reduced blood flow (ischemia) to an area of the heart. These changes may be clinically significant in certain patient populations, specifically patients who have had heart attacks. Changes in this portion of an ECG waveform occur slowly, can be asymptomatic (produce no pain or discomfort), and are often hard to detect by the user (physician or nurse). By providing this functionality in a patient monitor, we aim to supply the user with an early warning that one of these episodes of silent ischemia is occurring.

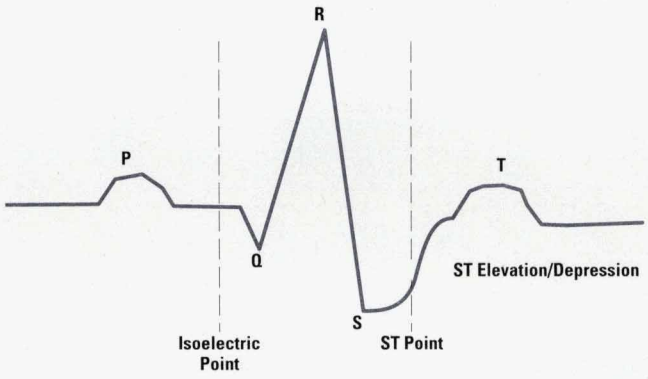


Fig. 1. ECG for one cardiac cycle (one heart beat).

Using HP-SL

As described in the article on page 24, HP-SL is a notation for describing systems and components in an abstract yet precise manner, allowing a user to focus attention on what a system should do and defer details relating to how to build the system until later stages of development. During this collaboration, we used HP-SL to create abstract yet precise descriptions (models) of various bits and pieces of the software to better understand the desired behavior early in the development process.

The software developed on this project continuously measures a portion of an ECG complex called the ST segment. Modeling the ECG wave at an abstract level makes it easy to understand and capture (document) essential properties of an ECG wave, properties that must exist for valid ST measurements to be made.

Modeling the ECG Wave

The ECG data that is input to the ST segment measurement algorithm is basically a stream of voltages (see Fig. 2), along with some configuration information such as:

- The voltage source (the lead)
- How much filtering has been done on the signal (the bandwidth)

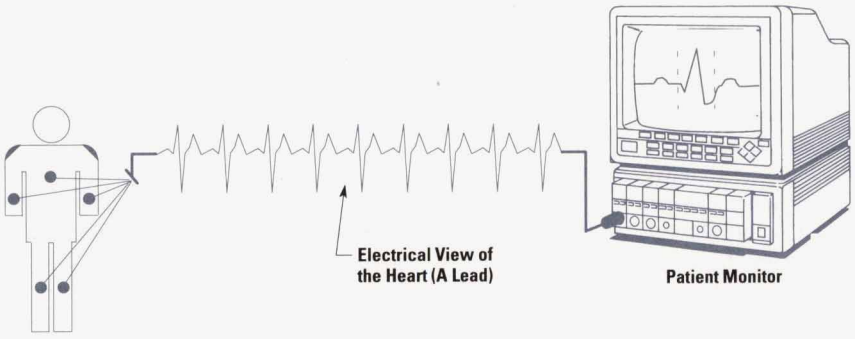


Fig. 2. Setup for making ECG measurements, showing the stream of voltage signals from one of the ECG leads.

- Wave content information such as which voltages represent usable patient information and which are invalid (the validity)
- Time.

The ST algorithm uses this incoming ECG data and other information to find and measure the ST segment of an ECG wave. If an ECG wave is measurable, an ST measurement will be produced. To capture the notion of measurability it is helpful to be able to look at the problem at an abstract level without concerns about resource limitations. ECG data is received as a sequence of wave samples that includes all the information mentioned above. Therefore, an ECG wave sample can be viewed as the cross product of these elements, and can be modeled using an HP-SL record as:

```
type Wave_sample ≜
        ⟦wave_sample ▷
            (voltage : Ecg_voltage,
            validity : Validity,
            lead : Lead,
            bandwidth: Bandwidth,
            time : Time)
        ⟧
```

An ECG wave can then be modeled simply as a sequence of wave samples:

```
syntype Ecg_wave ≜ Seq Wave_sample
```

Now the notion of measurability can be specified. An ECG wave is measurable if all of the wave samples are valid, have a specific bandwidth (0.05 Hz high-pass), are derived from the same lead, and are continuous in time. Using HP-SL this property can be expressed as a relationship involving the elements of a wave sample.

```
reln measurable : Ecg_wave
is measurable (ecg_wave) ≜
        ( ∀ (wv_sample1 ∈ elems (ecg_wave)).
            validity(wv_sample1) = valid
            ∧
            bandwidth(wv_sample1) = hz_point05
            ∧
            ( ∀ (wv_sample2 ∈ elems (ecg_wave)).
            lead(wv_sample1) = lead(wv_sample2))
            ∧
            continuous (ecg_wave))
```

The final relation, called continuous, specifies the requirement that the wave segment must be a continuous stream of wave samples (no gaps). This relation simply states that if two wave samples occur one after the other in sequence, the time associated with the second wave sample is one time unit after the time associated with the first sample. With HP-SL this notion is expressed as:

```
reln continuous : Ecg_wave
 is continuous (ecg_wave) ≜
      ( ∀ (i ∈ inds (ecg_wave), j ∈ inds (ecg_wave)) ·
          j = i + 1 ⇒
            time(ecg_wave(j)) ⇒ successor(time(ecg_wave(i))))
```

The HP-SL operator inds returns the indexes of a sequence.

In our products an ECG wave is typically implemented as a stream of data that consists only of voltages. Because of resource limitations (CPU time, RAM), the other information (validity, lead, etc.) is input via separate data streams where the data is updated less frequently. This works fine in an implementation if the developer knows what the relationship is between the streams of voltages and other data, so that changes in validity, for example, can be associated with the correct voltages. Problems arise if the developer hasn't learned, or forgets, these implicit relationships.

The model shown above makes these relationships explicit. For example, the requirement that all wave samples be derived from the same lead has been made explicit and available for review. By modeling the problem at an abstract level, implicit relationships can be made explicit. HP-SL provides a framework for making important properties explicit and increases the chance that an important property will be maintained across different implementations.

A Problem Uncovered

The process of understanding and specifying the notion of measurability uncovered a problem in an existing product. An ST measurement is actually made on a beat (see Fig. 1). A beat is a portion of an ECG wave that represents the electrical activity present during one cardiac cycle (one heartbeat). When a measurement is made, the corresponding beat is displayed to the user for review.

In one case the requirement that all of the wave samples that make up a beat be valid was not met. If a certain mixture of valid and invalid samples was present, the beat would mistakenly be considered measurable. The result was benign (the ST measurement was labeled invalid), but it was possible to display a strange looking beat

for a short amount of time. Our effort to capture a precise, abstract notion of measurability made this hidden problem visible for the first time.

Focus on Precision

One benefit that we noticed from our introduction to formal specification was that an emphasis on precision started to pervade our development process. We found that uncovering potential problems was pushed to an earlier stage of the development process than would have been the case had we not been focusing as much on precise specifications.

The primary function of a patient monitor is to transform the electrical signals coming from a patient into information that is useful to clinicians. These electrical signals are transformed into streams of digital samples representing a wave. This wave is analyzed and a numeric value is produced that represents a piece of the information contained in the wave, such as heart rate derived from an ECG wave, or blood pressure derived from a blood pressure wave. This numeric value is called a parameter.

In our case we needed to analyze three separate ECG waves and produce three separate ST measurements. We needed to give the user control over the ST measurements both as a group (e.g., turn the ST measurement for all the ECG waves on or off) and independently (measure the ST only on the second ECG wave). This requirement for both separate and combined controls was met through the use of a multichannel parameter. The ST parameter has three channels (ST1, ST2, ST3), with one ST value for each channel.

The patient monitor also contains data management software that provides the user with a trend of changes in the ST measurements. Depending on the on/off state of the ST parameter and channels, ST values are stored or rejected by the data management software. To communicate this information, the ST software needs to maintain two fields in a message, a parameter on/off field and a channel on/off field. These two fields were defined as Boolean fields within a larger data structure. The associated textual description was:

> Parameter ON/OFF: Parameters may be switched off from a central task window ... This has no effect on the internal functionality ...
>
> Channel ON/OFF: Channels may be switched on and off, but the effect is as with Parameter ON/OFF: Processing remains unaffected ...

We felt that our introduction to formal specification helped us quickly notice a potential problem, specifically the lack of an invariant in this description. HP-SL supports the specification of invariants on data that is being modeled, and when used in a data type definition, an invariant defines a relationship that must hold for all instances of that data type. A quick look at the description of the data structure containing the above fields shows that there is little discussion of any relationship between the parameter on/off fields and the channel on/off fields.

We expected to see an invariant indicating that if the parameter was in the off state then the channel must be in the off state. The lack of this invariant raised a warning flag. It turned out that the data management software received all three channels of ST measurement information, but used only the parameter on/off field to determine whether or not to store the information. When we turned a channel off but left the parameter on (a reasonable scenario from our point of view) the data management software would display an ST trend with invalid data, since no ST measurements were being made while the channel was in the off state. We were able to resolve this problem through discussions with the trend software developer long before the product was released to customers.

Similar ambiguous specification problems were encountered in a related area—alarm handling. There was no clear description of the expected behavior of multiple alarms on multiple channels of a single parameter. As a result we developed software that we thought behaved reasonably. After several unsuccessful attempts at making our alarm behavior match the expected alarm behavior (the behavior of the existing alarm handling software), discussions with the alarm software developer again led to resolution of the problem before product release.

If a description is ambiguous then multiple users of that description can easily have different interpretations. If a developer can provide a single, precise description (a model) of what is to be implemented, then there is at least an opportunity for the adequacy of the model to be tested through reviews, and ambiguities cleared up at the earlier stages of a project. The single, precise model may be wrong, but at least multiple reviewers can focus on the same wrong model instead of making incorrect assumptions about an ambiguous model.

Issues

Any new process, no matter how great the benefits, will encounter some difficulty in gaining acceptance or reaching the point where the benefits become real and tangible. Introducing formal specification is no exception. The issues raised fall into two categories: learning and applying formal specification, and introducing a new methodology into an existing software development process.

Learning and Applying Formal Methods. The issues related to this category included:

- Differing learning curves for reading versus applying the modeling tools provided by HP-SL.

This problem is akin to reading versus speaking a foreign language. If the words and syntax have already been put together by someone else, it is not difficult to follow most of what is written down. It is much more difficult to find the correct words and apply the correct syntax on one's own. A similar learning curve must be overcome in becoming proficient at creating models using HP-SL, and it involves practice and making mistakes. Mistakes are not easy things to figure into a schedule.

At the start of this collaboration the AMG gave a one-week HP-SL course to our project team and software engineers from other projects. The goal of the course was to introduce formal methods and HP-SL, and to have all participants feel comfortable reading HP-SL specifications.

This goal was reached. After the five-day course most participants felt comfortable reading HP-SL. But most also felt that more practice was needed to feel comfortable writing HP-SL specifications.

- Wondering if we could get the benefits of formal specification without learning another language or being so formal.

  This issue was raised frequently by ourselves and others during this collaboration. As mentioned earlier, we felt that the focus on early problem analysis had both a direct and an indirect impact on our development effort. For example, the problem discussed earlier in which the property of measurability was not met was found as a direct result of creating a precise specification. The second problem, the ambiguous parameter and channel descriptions, was quickly noticed as a result of our better understanding of what it means to create a solid specification. Even though we are still in the learning stage, we feel that we need to master the use of this rigorous approach to software development before we can make the best judgments about where added rigor can be most productive.

- Using formal specification techniques to capture the essential behavior of an existing software implementation.

  We found it to be very difficult to use formal notation to document the behavior of an existing implementation and felt that it was one of the least productive ways for us to make use of formal specification. We attempted this for software that was being ported from an existing product. Although this effort did in fact point out an existing problem, the resulting specification was not very abstract and ended up being focused on behavior that was a result of our specific implementation rather than on behavior resulting from a more abstract view of the problem.

- Understanding where increased formality is appropriate and where it is not.

  We found that deciding where to use formal specification was to some extent a judgment call. Our projects are typically not neat, tidy projects in which all of the requirements are precisely known at the start and each software engineer has a clean sheet of paper from which to begin development. Many of the medical algorithms and associated behaviors (e.g., alarm handling) need to be reimplemented (with enhancements) in new products, and there is often software in another product (e.g., databases, data review screens) that can be leveraged.

  For example, to create a display for a new measurement in the patient monitor, a developer typically uses existing display software as a template, a case in which there doesn't appear to be much of an underlying model to capture. On the other hand, we did find that the time spent modeling the notion of measurability was useful. In this case, there appeared to be a significant difference between the underlying model and the implementation. Our software development efforts are best termed software reengineering, and in this setting, we feel that formal specification needs to be mastered and used as a tool at the discretion of the developer.

Introducing a New Technology. The issues related to this category included:

- The effect on project schedule.

  The use of this level of rigor for software development was new to us, and we devoted a lot of effort learning how to use formal specification for our tasks. At the start of the project we scheduled time for taking an HP-SL course and workshop. In addition, we lengthened the preimplementation stages of the schedule to account for what we thought was the additional time required to learn and apply formal specification. In retrospect, we underestimated how much time it would take to get comfortable with this tool and overestimated the number of portions of the software task that we thought we could specify. Limiting our efforts to more manageable-sized portions of the task is one way we could have limited the potential schedule cost.

- The need for up-front commitment of resources.

  On this project, as on most of our projects, two very precious resources are time and money. The additional time in a project schedule allotted to learning a new software development method must be accepted by all parties. Also, the transfer of this technology involved a collaboration between geographically separated groups. To collaborate means to work together, and this, despite modern systems of electronic communication, requires some face-to-face time. In our case travel was a necessary part of such a collaboration.

- The difficulties in measuring the benefits of formal specification.

  Improvements in software quality can be difficult to measure based on the metrics from a single software project. The size of the project, design and code complexity, skill sets of the personnel involved, and time allotted to the project can all affect the metrics used to measure the benefits of a software process change.

  In Table I the metrics for our project are compared to a project of similar complexity. Our metrics were good, and, although there are a number of ways to explain the data (e.g., that defect density is proportional to test hours) we believe that the focus on up-front problem analysis encouraged by learning and using a formal notation played a significant role in achieving a high-quality product.

Table I
Comparison of Project Metrics

Project	KNCSS*	Test Hours	Defect Density**
Our Project	16.1	20.3	0.06
Project X	81.8	61.2	0.20

*Thousands of noncomment source statements.

**Defects per KNCSS.

Conclusion

Many challenges must be faced in a collaboration to introduce a new methodology into an existing software development process. The biggest change we would make is to have all parties view the collaboration as part of the project, not something extra that can be done if time permits. Only one schedule should exist and it should include allowances for the collaboration. Only if the effort is given this importance and support can the people involved make it a true success.

We used this collaboration as a vehicle for learning more about the benefits and costs of using formal specification. In practice it was very difficult to balance the introduction of a new method with the need to produce a relatively simple software enhancement in a short amount of time. We are convinced that there is much to be gained by using a formal specification language like HP-SL to capture the essential behavior of the software machines that we design and build, and are continuing to use this methodology in our development efforts.

Acknowledgments

We gratefully acknowledge the continued support and contributions from Fran Michaud, George Diller, and Wolfgang Krull from HP's Waltham Division, and Mike Diss, Sally Jubb, Tony Rush, Paul Harry, and Ray Crispin from HP Laboratories in Bristol, England.

Bibliography

1. J. Woodcock and M. Loomes, *Software Engineering Mathematics*, Addison-Wesley, 1988.
2. T. Rush, et al, *Case Studies in HP-SL*, Software Engineering Department, HP Laboratories Bristol, HPL-90-137, August 1990.
3. C. B. Jones, *Systematic Software Development Using VDM*, Prentice-Hall International, 1986.
4. A. Davis, *Software Requirements Analysis & Specification*, Prentice-Hall Inc., 1990.
5. T. Denvir, *Introduction to Discrete Mathematics for Software Engineering*, MacMillan Education Ltd., 1986.
6. D. Garlan and N. Delisle, "Formal Specifications as Reusable Frameworks," *Lecture Notes in Computer Science*, Vol. 428, Springer-Verlag 1990.

Formal Specification and Structured Design in Software Development

HP-SL history specifications and techniques from structured analysis are used to create a formal specification for a critical portion of the code for a medical instrument.

by Judith L. Cyrus, J. Daren Bledsoe, and Paul D. Harry

The cardiology business unit at HP's McMinnville Division is responsible for producing medical instruments, some of which have life-critical functionality and require a high degree of reliability. These instruments are used in a high-tension environment by medical personnel who are not necessarily computer literate and do not use the instruments on a daily basis. Our project team (from the cardiology business unit) is responsible for the development of one of these life-critical instruments.

Previous generations of the product are viewed as having defect-free software, but there is no formal way to verify this belief. The code in these earlier instruments was written in assembly language and is tightly coupled with the hardware. Also, the code is considered to be hard to understand and is not reusable. To remedy this situation we established the following goals for the new software in the product:

- Produce defect-free, safety-critical software with fewer debug cycles
- Create software that can be reused in the current product family and in future products
- Write unambiguous documentation that describes the safety-critical functionality of the product
- Perform a validation process that compares the implementation against the specification
- Meet or exceed regulatory requirements for safety-critical products.

This paper describes our experiences with using formal specification techniques to help implement a safety-critical portion of the embedded software system for the instrument.

Why Formal Methods?

During the project investigation stage, formal methods were seen as an aid toward meeting our software goals. We were convinced that we should use the best available methods to achieve the high reliability needed for our product. We also saw formal specification as a way to ensure that product requirements were well-understood by our current project team and by future development teams or maintainers. The applied methods group (AMG) from HP's Bristol laboratories provided us with information about their work with a formal specification language called HP-SL (HP Specification Language) and offered to collaborate with us on the project. We saw the collaboration as a way to establish expertise in formal methods within our division.

The AMG had previously collaborated with another group at our division to use formal notation for a small part of the software for an another product. Although this was done on a very small scale with limited success, there was enough positive impact to influence management approval for our decision to use formal methods. This was a critical factor in successfully using formal methods because we felt that we needed to include time in the project schedule for using formal specification techniques.

After deciding that we wanted to use formal methods in our development process, the first thing we did was to establish a collaboration plan between the HP McMinnville project team and the consultants from the AMG in Bristol. This plan defined the roles of the collaboration team members and established a modified product life-cycle plan that provided for the anticipated front-end loading on the project schedule. One McMinnville engineer and one AMG engineer actively engaged in the safety-critical software development effort. Other collaboration team members participated in support activities, primarily serving as reviewers.

The Formal Specification

The team decided that it would be too ambitious to attempt to specify the entire software system formally. Therefore, the system functionality was divided into safety-critical and nonsafety-critical subsystems. Fig. 1 illustrates this division. The safety-critical code is located in the safety-critical controller and occupies about 16K bytes of ROM.

Based primarily on the product's external specification (ES),* the collaboration team produced a complete formal specification of the safety-critical part of the software. The formal specification provided a process model of the system, specifying the relationship of inputs and outputs over time. This model uses the HP-SL history specification (see article on page 40), which associates data values with time, thereby capturing timing constraints and specifying a data object's value for all time. This allows comparisons between data objects at any particular time, and

*The external specification is a natural language description of the product requirements.

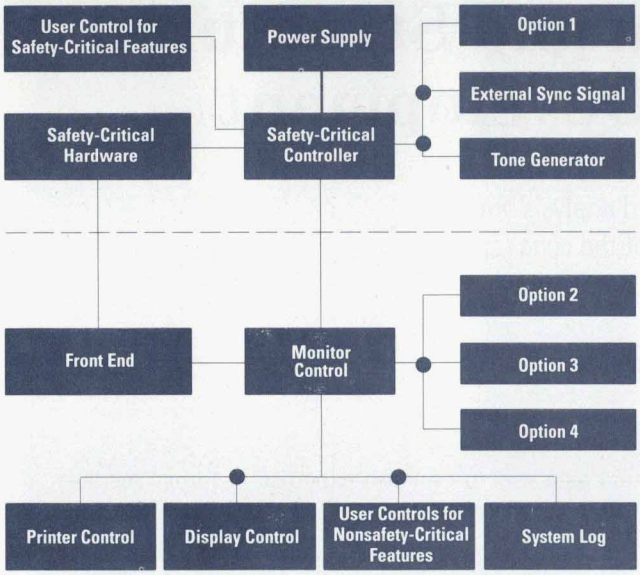


Fig. 1. System architecture showing the separation between safety-critical and nonsafety-critical functionality.

provides a history of what has happened before so that new data values can be derived from previous values.

The specification is illustrated by a variant of data flow diagrams. These diagrams look much like traditional structured analysis data flow diagrams,[1, 2] but the meaning of some of the symbols is changed to better illustrate the formal specification. An example of one of these diagrams is shown in Fig. 2. The data flows between processes correspond to HP-SL histories, and the dashed lines, which indicate control data in traditional structured analysis data flow diagrams, are used to indicate optional data which is only present in some product family members.

Early in the project, we decided to use two views of data to support a need for concreteness at the external level and abstraction at an internal level. The external view represents hardware dependencies such as the register bits that hold voltage information. The internal view provides a hardware independent representation of data—for example, a data structure that just holds voltage values regardless of the source or destination. The internal view, or core, specifies instrument functionality that is common to all of our product family members. This two-level view allowed us to separate and more abstractly specify the core of the system in a way that isolates it from the impact of hardware changes and makes it easier to reuse. Specifying the external view in a concrete way enabled us to correlate our design with the hardware design specification.

The formal specification includes an HP-SL data model of the external and internal views, HP-SL conversion processes which specify how the internal view of the data is derived from the external view, and a complete HP-SL specification of the system core process. The simplified top-level data flow diagram shown in Fig. 2 shows this data view and the conversion processes that convert from the external view to the abstract internal view. Data flows (histories) coming into and going out of the diagram around the edges illustrate external interfaces. The six bubbles around the outside illustrate the conversion processes with the internal data flowing into and out of the core process.

A preliminary formal review raised many requirement issues. Initially we thought that the product's external specification would be a sufficient source on which to base the formal specification. It turned out that this left many requirement issues ambiguous or undefined and did not provide enough information about the hardware and

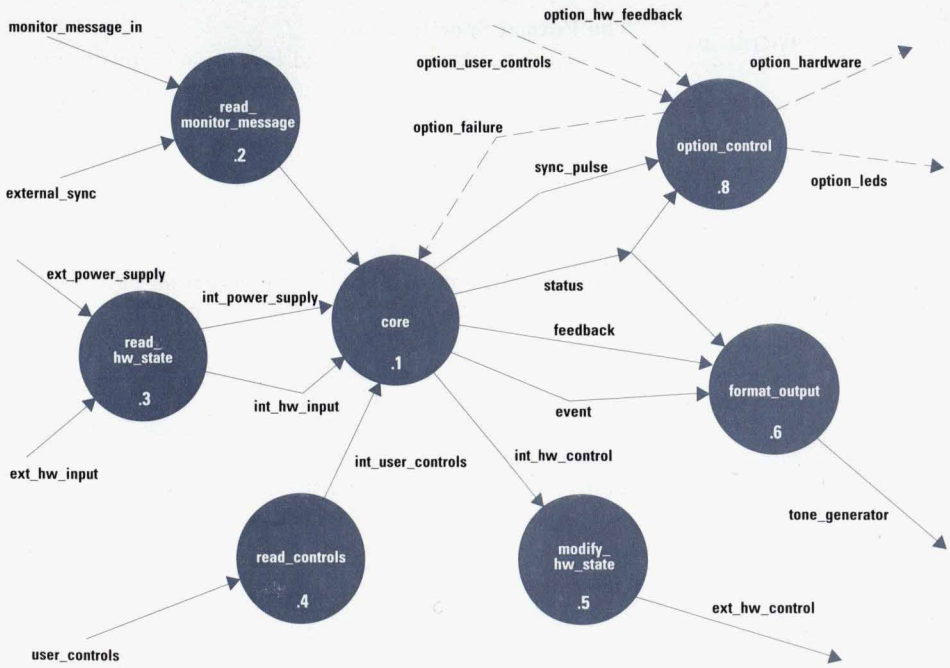


Fig. 2. Data flow diagram showing the processes involved in transforming the external view of data into the internal, or abstract view.

timing to complete the specification. Following the preliminary review, project team members were asked to provide timing diagrams and hardware and software interface documents, which would otherwise have been required later in the project. Only when these were available could the formal specification be completed.

Implementation Route

One of the major benefits of using an HP-SL specification structured as a data flow diagram is that the traditional route from structured analysis to structured design can, with modifications, be used. The data flow diagrams that illustrate the HP-SL specification drive the basic layout of structure charts.

Fig. 3 provides a simplified comparison between portions of the traditional structured analysis to structured design implementation route and the formal process specification implementation route. In both approaches the process specifications define the system requirements and the module specifications define the design and implementation of the system.

In the traditional approach, informal process specifications define primitive processes (processes that are not decomposed into further processes). Composite processes, and ultimately entire systems are defined by combining the primitive processes according to the data flow diagrams. Module specifications describe the functionality of the individual routines that implement the system.

In the HP-SL approach, a process specification is a precise description of the transformation of incoming data into outgoing data. The modified data flow diagrams define how the process specifications are combined to form the process specifications of higher-level processes. As in the traditional approach, module specifications define the implementation routines.

This systematic development route preserves traceability between modules of the implementation and processes of the specification. Traceability permits verification of the implementation against the specification, and also allows test plan designers to make the best use of the specification in defining test cases. The following steps are taken to implement this systematic development route.

1. Start with an HP-SL specification and the accompanying data flow diagram. Fig. 4 shows the data flow diagram for one of the subsystems in the product. Remember, dashed lines indicate optional data flows, which exist only in systems that have this option installed.

2. Treating the data flow diagram from step 1 as a classical structured analysis data flow diagram, transform it into a hierarchical structure chart. Fig. 5 illustrates a structure chart for the example process. In this example, the second-level modules correspond to the processes of the data flow diagram, except for the process derive_option_failure, which has been moved to a lower level for a more efficient implementation. The data couples shown in Fig. 5 represent entire histories.

3. No implementation can afford to pass entire histories between modules. A reduced design must be produced in which each history is replaced by just the current value.

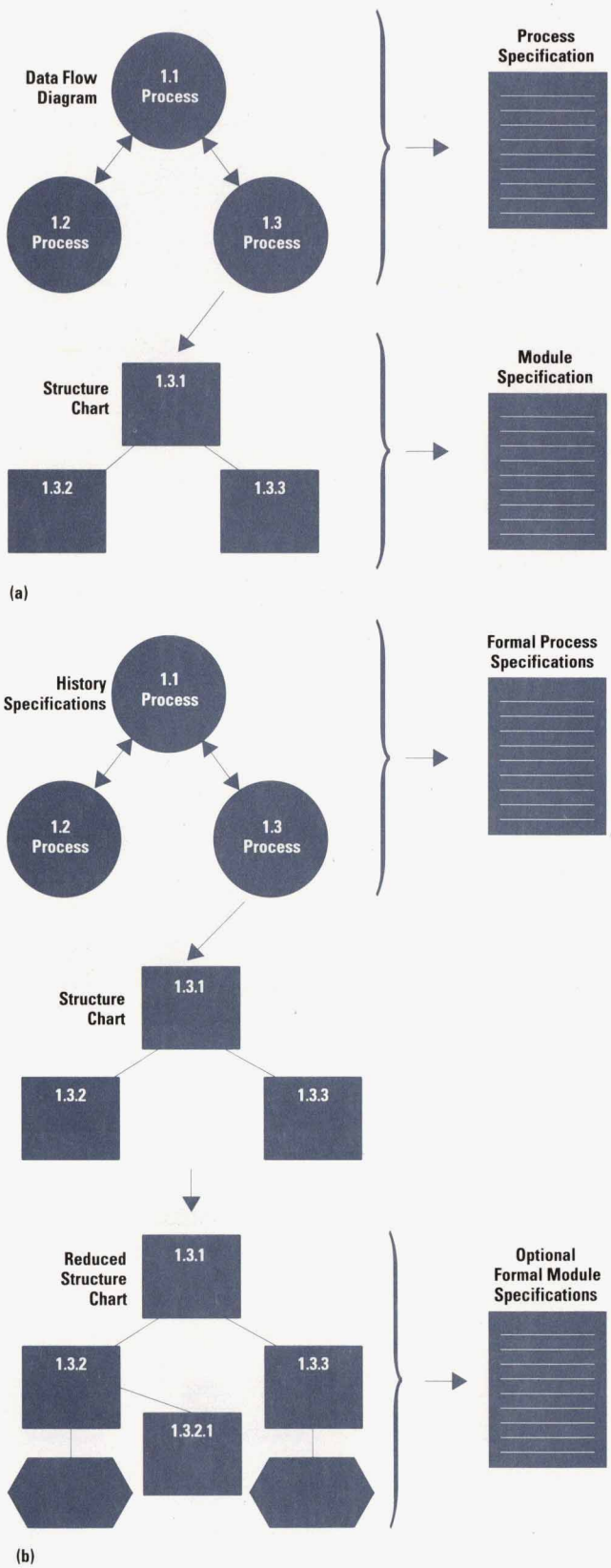


Fig. 3. The production of module specifications. (a) In the traditional SA/SD approach they are produced after structure charts are defined. (b). In the formal specification implementation, the process specification can usually replace the need for module specifications.

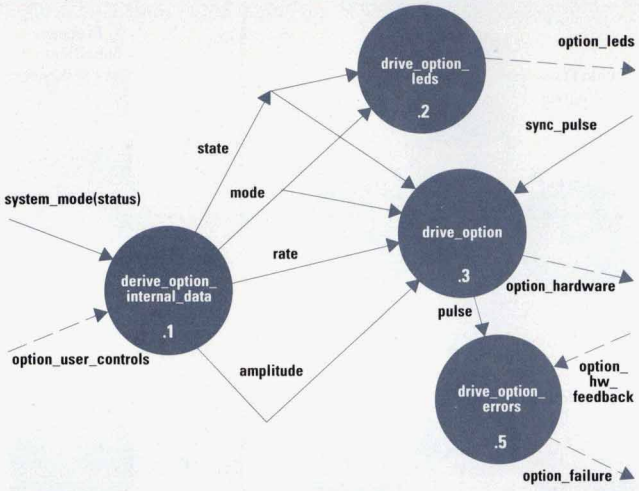


Fig. 4. Data flow diagram for example subsystem.

Any historical information required must be explicitly stored in local stores (see the hexagon-shaped boxes in Fig. 6). Notice that two local stores have been added to module time_pulse. One keeps track of the time since the last pulse was fired and the other keeps track of the time since the last external pulse was detected.

4. The structure chart is optimized. For example, modules are moved to different levels in the hierarchy (see the reduced structure chart in Fig. 3b).

5. If necessary, produce module specifications. Many times modules are closely related to the process specifications, thus eliminating the need for module specifications because the process specifications are sufficient. However, if the optimization performed in step 4 is extensive, optional module specifications may be required (see Fig. 3b).

6. Code is written from the structure charts, the HP-SL process specifications, and if any were generated, the optional module specifications.

Implementing a Process Specification

The data flow diagrams and structure charts shown in Figs. 4, 5, and 6 provide a graphical representation of the software being developed using the formal implementation steps described above. This section describes the development of the HP-SL process specification for the process time_pulse, and the module that implements this process (labeled time_pulse_m in Fig. 6). The history and event

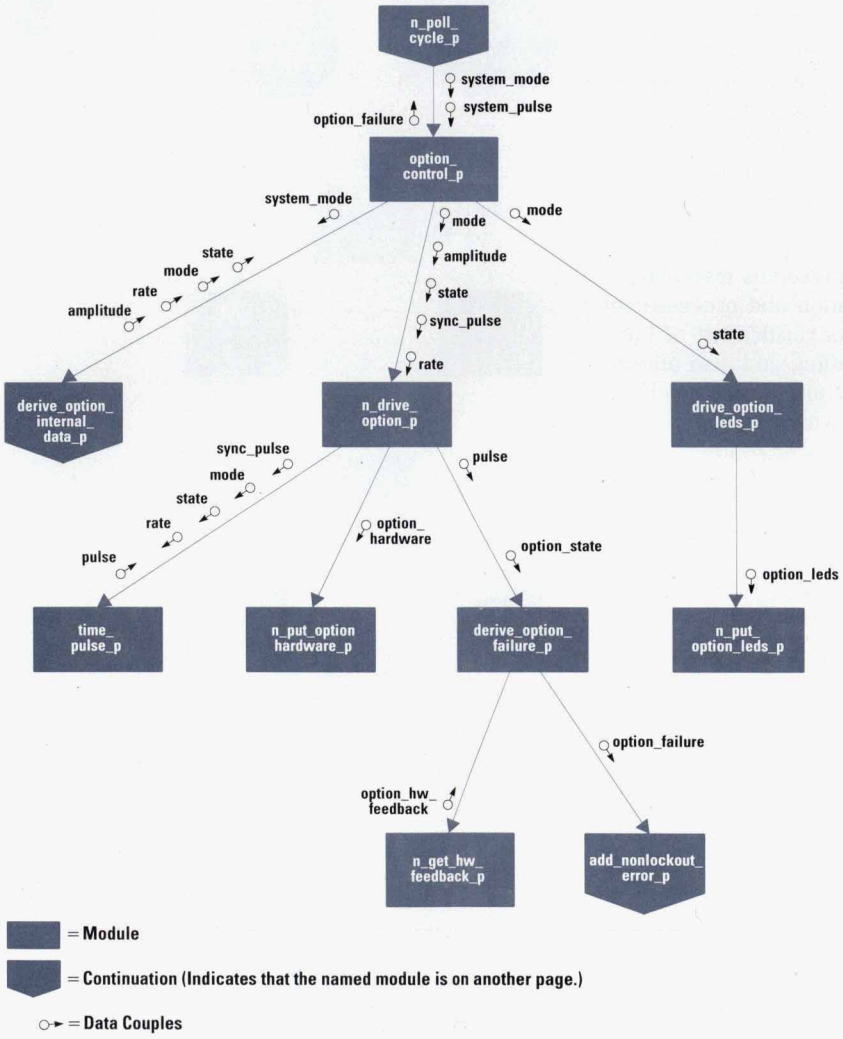


Fig. 5. Structure chart using histories. Data couples are histories.

specifications described in the article on page 40 are used to develop the HP-SL specification for this module.

Pulses. Before considering the time_pulse process, we first consider pulses and the definition of an overdue pulse. A pulse is a dataless event occurring within a specified time interval. A dataless or single-valued item can be modeled by the type Unit. Pulses can then be modeled by an event history of units.

$$\text{syntype } \text{Pulse}_h = \text{Hist}_e(\text{Unit})$$

If we have a history pulse_h of type Pulse_h, we can test a pulse at time t with the expression pulse_h @? t, which is true if and only if a pulse occurred at t.

A pulse is overdue if, from some time t_start to the current time now, there have been no pulses, and the system has been in state running. This is formalized by the expression:

$$\begin{aligned}&(\ \forall\ t \text{ sat } t\_start \le t \ \wedge\ t < now\ \cdot\\ &\qquad \neg\, \text{pulse}_h\ @?\ t\\ &\qquad \wedge\\ &\qquad \text{state}_h\ @\ t = running\\ &)\end{aligned}$$

The time t_start is fire_interval_msecs (time between pulses) earlier than current time, or t_start = now − fire_interval_msecs.

As in the raise_alarm process (see article on page 40), we must consider the behavior just after startup. When now < fire_interval_msecs a pulse cannot be overdue. This behavior can be specified by an existential quantifier, ∃ (exists), which tests whether t_start is earlier than the startup time. Combining the test for an overdue pulse with the test to verify that t_start is earlier than the startup time, we get:

$$\begin{aligned}&\text{fn pulse\_overdue: } \text{Hist}_e \text{ Pulse} \times \text{Hist}_s \text{ State} \to \text{Bool}\\ &\text{is pulse\_overdue}(\ \text{pulse}_h,\ \text{state}_h\ )\\ &\triangleq\\ &\text{let}\\ &\quad \text{val now} \triangleq \text{end}_s(\text{state}_h)\\ &\quad \text{val start\_time} \triangleq \text{start}_s(\text{state}_h)\\ &\quad \text{val fire\_interval\_msecs} \triangleq 1000 / \text{rate}_h\ @\ now\\ &\text{in}\\ &\quad (\exists\ t\_start = now\ -\ fire\_interval\_msecs \text{ sat } t\_start \ge\\ &\quad start\_time\ \cdot\\ &\quad (\forall\ t \text{ sat } t\_start \le t\ \wedge\ t < now\ \cdot\\ &\qquad\qquad \neg\,(\text{pulse}_h\ @?\ t)\\ &\qquad\qquad \wedge\\ &\qquad\qquad \text{state}_h\ @\ t = running\\ &\quad)\\ &)\\ &\text{endlet}\end{aligned}$$

which returns a TRUE value when a pulse is overdue.

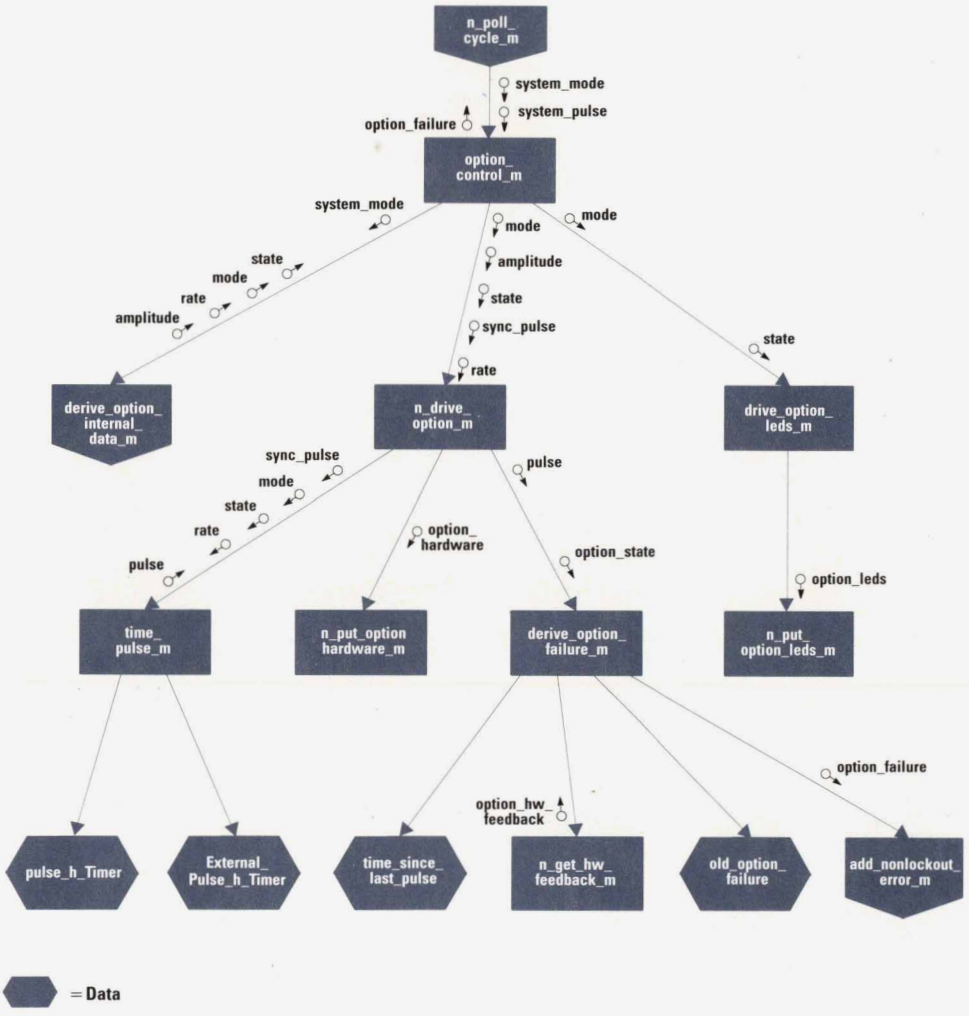


Fig. 6. Structure chart without histories. Local stores preserve necessary historic data.

Time Pulse Process. The time_pulse process generates pulses, and there are two pulse generation modes: deferred and fixed. In fixed mode pulses are generated periodically and in deferred mode pulses are generated only if an external source of pulses has failed. The process has four inputs: system state, pulse mode, the required rate of pulse generation (which determines the pulse interval), and the external pulses. There is one output—pulses generated when a pulse is overdue in either mode.

The process specification for time_pulse is of the form:

reln time_pulse:
    ($Hist_s$ State × $Hist_s$ Mode × $Hist_s$ Rate × $Hist_e$ Pulse) ×
    ($Hist_e$ Pulse)
is time_pulse(($state_h$, $mode_h$, $rate_h$, $external\_pulse_h$), $pulse_h$)
≙
...

The occurrence of an output pulse can be detected by the expression $pulse_h$ @? now. However, we want to base pulse generation not only on when an output pulse occurs, but also when the pulse does not occur. This can be achieved with the operator ⇔, read as "if and only if."

In fixed mode, an output pulse is generated if and only if there has been no output pulse for more than a given time, that is, if the output pulse is overdue. This property can be formalized by the expression:

$pulse_h$ @? now
⇔
pulse_overdue($pulse_h$, $state_h$)

In deferred mode, there is an additional constraint. Not only must there have been no output pulse, but there must also have been no external pulse.

pulse_h @? now
⇔
(pulse_overdue($pulse_h$, $state_h$) ∧ pulse_overdue($external\_pulse_h$, $state_h$))

Which test applies depends on the current mode ($mode_h$ @ now).

Process Specification. The following is the final specification for the time_pulse process.

```
 1: reln time_pulse:
 2:   (Hist_s State × Hist_s Mode × Hist_s Rate × Hist_e Pulse) ×
 3:   (Hist_e Pulse)
 4: is time_pulse( (state_h, mode_h, rate_h, external_pulse_h), pulse_h)
 5: ≙
 6: let val now ≙ end_s(state_h)
 7: in
 8: pulse_h @? now ⇔
 9:       cases mode_h @ now of
10:         case [[ fixed ]] then
11:             pulse_overdue(pulse_h, state_h)
12:         case [[ deferred ]] then
13:             pulse_overdue(pulse_h, state_h)
14:             pulse_overdue(external_pulse_h, state_h)
15:       endcases
16: endlet
```

This relation specifies that a pulse occurs (line 8) in fixed mode (line 10) if the interval since the previous generated pulse (line 11) reaches the interval determined by the rate setting in the pulse_overdue function. A pulse occurs in deferred mode (line 12) if both the interval since the previous generated pulse (line 13), and the interval since the previous external pulse (line 14) reach the interval determined by the rate setting in the pulse_overdue function.

The test for mode and overdue pulses (lines 9-15) can be rewritten as:

pulse_overdue($pulse_h$, $state_h$)
∧ ($mode_h$ @ now ≠ deferred ∨ pulse_overdue($external\_pulse_h$, $state_h$))

In the C code implementation described below, this decision structure is used directly on lines 25 to 30.

Implementation. The following is the C code for implementing the time_pulse function.

```
 1: #include "ic_global.h"
 2: typedef enum { PULSE, NO_PULSE } type_pulse ;
 3: typedef enum { FIXED, DEFERRED } type_option_mode;
 4: typedef enum { RUNNING, STOPPED } type_option_state;
 5: extern unsigned_16 rate_table[];
 6: type_pulse time_pulse(external_pulse, mode, state, rate )
 7: type_pulse external_pulse;
 8: type_option_state state;
 9: type_option_mode mode;
10: unsigned_8 rate;
11: {           /*** Begin Function ***/
12: static unsigned_16 external_pulse_h_timer;
13: static unsigned_16 pulse_h_timer;
14: type_pulse pulse;
15: /* Reset the timers whenever the option is not running.*/
16: That way the timers will never expire and no pulses will be fired*/
17:  if (state != RUNNING)
18:  {
19:   external_pulse_h_timer = 0;
20:   pulse_h_timer = 0;
21:   pulse = NO_PULSE;
22:  }
23:  else      /* option is running */
24:  {
25:   pulse = (( pulse_h_timer >= rate_table[rate]) /* pulse overdue */
26:       &&          /* and if not in deferred mode*/
27:       (( mode != DEFERRED) /* the ext. pulse must also be overdue
*/
28:         || (external_pulse_h_timer >= rate_table[rate]))
29:      ? PULSE
30:      : NO_PULSE;
31:   if (pulse ≙ PULSE) pulse_h_timer = 0;
32:   else if (pulse_h_timer <= rate_table[rate:) pulse_h_timer++;
33:   if (external_pulse ≙ PULSE) external_pulse_h_timer = 0;
34:   else if (external_pulse_h_timer <= rate_table[rate])
                        external_pulse_h_ timer++;
35:  }       /* end option is running */
36: return(pulse);
37: }
```

The two calls to pulse_overdue are replaced by two timers, external_pulse_h_timer and pulse_h_timer (lines 19 and 20). They are reset to zero whenever the state is no longer running (lines 17 to 22), or the corresponding pulse occurs (lines 31 and 33), otherwise they are incremented (lines 32 and 34).

In the specification world we typically do not worry about size limits. However in an implementation, size limits are critical. If we repeatedly increment the timers they could overflow. Fortunately we are not interested in how big the timers get once they are beyond the overdue time. Thus the code stops incrementing the timers once they reach the rate table limit (lines 32 and 34).

Impact on Project

The decision to use a formal notation for the safety-critical software had a major impact on the project schedule and the way in which we performed our implementation. Because of the learning curve we found that front-end loading the project schedule was necessary to use formal methods successfully for the first time. In our case the steep learning curve was offset by support from the HP Bristol team. Thus, the overall impact on the schedule turned out to be minor. The benefits of a more complete external specification (ES), early requirements decisions, complete interface specification, and a good software design paid back the time needed to produce the formal specification. In fact, the completeness of design made it possible for one engineer to code the entire safety-critical subsystem in less than the usual amount of time required to complete a subsystem of this type.

The impact formal specification had on our implementation is that it enabled us to identify and deal with problem areas early in the project, contributing both to the quality of the final external specification and to hardware design decisions. Specific areas of this impact include:

- To isolate the safety-critical software, we decided to use separate microcontrollers for safety-critical and nonsafety-critical tasks.
- The formal specification affected the structure and content of the ES. It exposed ES ambiguities and incomplete product definition. For example, the ES described normal system functionality, but did not define system behavior during abnormal situations, such as when multiple keys are pressed simultaneously. The specification forced us to deal with these issues.
- Problem areas in the formal specification identified the need for additional documentation (timing diagrams and hardware/software interface documents) early in the project. A timing issue that was mentioned in general terms in the ES was exposed by the specification as a hazard. These timing problems had to be better understood before the specification could be finished.
- The need to make firm requirements decisions for writing a formal specification that could be reused in follow-on products encouraged early definition of possible future systems and peripheral interfaces.
- Work on the formal specification helped new team members learn the product requirements by exposing areas of misunderstanding or ambiguity.
- The formal specification identified important test cases and boundary conditions. Some processes have preconditions or invariants that define corner cases, which had to be tested.
- The formal specification helped in dealing with the regulatory process by helping define and plan for potential hazards (e.g., interprocess communication failure).
- Software changes resulting from late hardware changes were easy to make and document.
- The goal of maximizing code reusibility for the safety-critical software was achieved.

Problem Areas

We did encounter a few problems during development. These can be classified as problems associated with the project development process and problems associated with using formal methods.

Problems we found associated the development process were:

- The external specification has a dual role in our division. It serves both as the natural language requirements document and as an input to the business decision at the investigation-to-implementation checkpoint meeting. This typically results in an ES that is not optimally suited to either, and in many cases does not contain sufficient detail on which to base the formal specification. This problem can be solved by producing, in addition to the business-needs-driven ES, a software requirements document that addresses functionality details, timing requirements, and hardware and software boundaries.
- Some interfaces were not tied down for a long time. This meant the formal specification for the interfaces had to wait, and sometimes this meant changing the core process when the interface was finally understood.
- The need for a concrete definition of the boundaries and interfaces when specifying only a part of the system was not initially recognized. Early availability of interface requirements would have greatly simplified the specification process.

Problems we encountered with using formal methods were:

- When we learned HP-SL, we learned notation, but not abstraction or modeling. It is very hard to stop thinking in programming terms. A better grasp of abstraction would have helped in getting the most benefit from the modeling power of formal notation. This comes with practice.
- Clever parts of the HP-SL specification were intellectually pleasing but not understandable by other members of the team or reviewers.
- Formal notation does not provide early estimates of code size and execution performance. This means that gross estimations (guesses) were used for ROM and RAM requirements—decisions that were needed early in the project. As it turned out, these estimations were remarkably accurate.

Conclusion

A number of factors were vital to the successful completion of the formal specification and its usefulness in our software development. The most critical factor was that the decision to use formal notation was made early in the project. Experience on other projects has shown that retrofitting abstractions into an existing product implementation is not very successful. A late decision would also have reduced the positive impact the process had on the external specification document. There were several key factors that contributed to our success with formal specification. Some of these include:

- The collaboration with HP Labs was crucial. The HP Lab team from Bristol spent some time with us to begin the specification and their participation in the product's HP-

SL review, which included both the hardware and software engineers from McMinnville, provided necessary HP-SL expertise.

- The diagrammatic overview of the specification provided both a visual way to see how the parts of the specification fit together and a more traditional view of the software which is useful when working with non-HP-SL readers.
- The division of the specification into an abstract reusable core and a less abstract, customized interface, saved rewriting the entire specification when hardware changes were made.
- The implementation route paralleling traditional approaches helped reduce the negative impact of introducing a new development process.

Formal specification has often driven the capture of product requirements and product design. Even hardware design has been influenced. The safety-critical nature of our product steered us to use a formal notation. However, the influences of the process on our project demonstrate that it provides benefits which are helpful to any project, and not just safety-critical projects.

References

1. T. DeMarco, *Structured Analysis and System Specification*, Yourdon Press, 1978.
2. S. Mellor and P. Ward, *Structured Development for Real-Time Systems*, Vol. 3, Yourdon Press, 1986, pp. 91-95.

Telecommunications Network Monitoring System

This system supervises any telephone network using the 2-Mbit/s CEPT primary rate interface and the CCITT R2 or #7 signaling system. It automatically collects and analyzes data on CCITT-specified and other parameters related to the calls flowing through the network nodes.

by Nicola De Bello, Giuseppe Mazzucato, Antonio Posenato, and Marco Silvestri

Telecommunication networks need to be monitored for several reasons. First, for planning purposes, the network manager has to identify the most effective locations for investment in new resources. To do this the manager needs to know the network areas where the traffic is most critical and the quality of service is low. Monitoring is also done to certify the quality level of the network as seen by normal users and, more often, by managers responsible for other connected networks. The third reason for monitoring is to identify faults and verify the behavior of the network in critical situations. All of these reasons are growing in importance, while telephone networks are becoming more complex and geographically extensive.

Network behavior is monitored by measuring some parameters related to the calls flowing through the network nodes. These parameters, in part specified by the CCITT, provide information on several aspects of network status. Some parameters, such as the number of telephone calls and their mean duration, are related to the traffic intensity and type. Others, such as the number of calls resulting in a congestion signal, provide a measurement of the quality of service offered by the network. The parameter most commonly used for this purpose is the answer-seizure ratio (ASR), defined as the ratio between the number of successful calls (answers) and the total number of call attempts (seizures). This ratio represents the probability that a user will be able to complete a call on any given attempt. Other parameters are related to network efficiency and use. In any case, to understand network behavior fully, it is very important to calculate the parameters separately for each of the paths in the network during the time of interest.

Monitoring System Requirements

Parameter measurements have to be made using instruments distributed throughout the network of interest. Then, to analyze the network from a global point of view, the raw data has to be correlated with network topology information.

Some network elements contain embedded instrumentation for monitoring, but it is difficult to use this information for monitoring all of the network. In fact, the elements of a telephone network are usually made by different manufacturers using various technologies, so the performance information that can be gathered is not homogeneous. Another point to note is that monitoring is not the main function of the network elements, so when one of them becomes busy performing its main function, it usually means that the measurements must be halted.

A monitoring system has to be very flexible. In fact, there are several different types of networks, from small private networks to large public ones, from local access networks to those covering very wide areas. Every network has its own problems to solve and every organization has its own way of resolving them. Also, inside an organization, there are several departments interested in different aspects of network behavior. A system for telephone network monitoring has to be adaptable to meet all of these situations.

Telephone networks are always evolving to meet their users' needs. It is therefore important that the monitoring system be able to grow with the network.

Monitoring is a distributed process, so it is more efficient to perform a greater part of the processing where the data is collected than to centralize the data for processing. This improves data transmission efficiency and the scalability of the system. The distributed approach also ensures uniform quality and format of the data across the whole network irrespective of network element types.

System Architecture

The HP E3500A network monitoring system, shown in Fig. 1, is composed of two main parts:

- Peripheral units, which are connected to the network and collect the data
- A central unit, which is a processing system that elaborates the data and provides a user interface to the system.

The HP E3500A network monitoring system is applicable to networks using the CCITT R2 and CCITT #7 signaling systems and the CEPT 2-Mbit/s primary rate interface. The peripheral units collect the data related to the network behavior by analyzing the 2-Mbit/s streams used to connect the network switching elements. This monitoring point has been chosen because it is a logical border be-

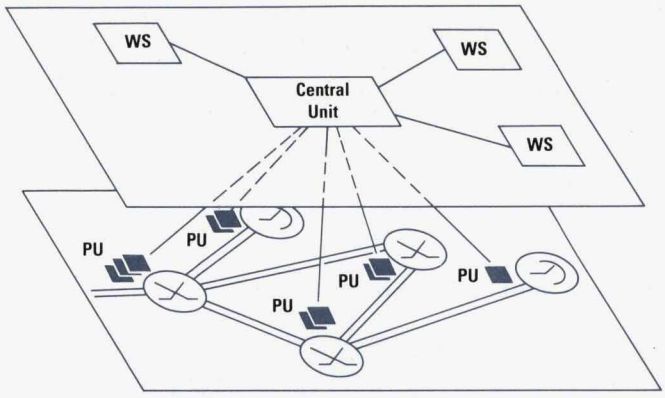


Fig. 1. HP E3500A network monitoring system architecture. The central unit is an HP 9000 Series 300, 400, or 800 computer. There are typically 2 to 6 users. The network symbols on the lower layer represent local and transit exchanges.

tween the "active" elements and it provides an external view of the network that is similar to the user's point of view. In addition, the 2-Mbit/s PCM digital link has been accepted as a standard in Europe and in many other countries around the world so there are fewer interfacing problems.

The peripheral unit analyzes the information on the digital link by monitoring the telephone calls flowing in the link. The peripheral unit stores a database record for each telephone call observed. This record contains all the information about the call that is required for calculation of the call parameters mentioned above. The simplified call record format is shown in Fig. 2. To perform its job, the peripheral needs to capture the signaling messages sent by the exchanges in both directions and recognize the service tones sent to the user. There are several international standard signaling systems and a number of country-specific and vendor-specific versions of these standards. The peripheral unit's hardware and software were designed to be able to work with different standards and make the actual signaling system used transparent to the upper levels. The data format is almost independent of the signaling system in use so it is easy to integrate the data coming from the different links.

The peripheral units are distributed around the network and are connected to the central unit using, typically, leased lines and modems. In the case of peripheral units located near the computer it is possible to make the connection using an RS-232 cable. If leased lines are not available it is also possible to use switched lines. To minimize the number of lines used to communicate with peripherals, several peripheral units can be connected together and managed by the central unit using a single communication line. Interconnection and control of the peripherals connected in this way are realized using the RS-485 serial bus and the ISO 1745 multipoint communication protocol.

Peripheral unit operations are completely controlled by the central unit. The central unit bootstraps the peripheral units with the appropriate application program, sends measurement commands, and monitors peripheral unit status. In general, the system manager can manage the measurement system completely from any of the terminals attached to the central unit. For maintenance purposes, a portable operator terminal is also available which can be connected locally to the peripheral unit.

Peripheral Unit

The number of PCM links to be monitored at each location depends on the organization of the network. Additionally, different signaling systems can coexist at the same measurement site, and particular functions may be required only at certain network nodes.

These facts led to a modular architecture for the peripheral unit, in which the required special functions are performed by optional boards. Speech signal processing is performed digitally to ensure a high degree of flexibility.

The peripheral unit is a multiprocessor device organized in a master-slave architecture. As Fig. 3 shows, the peripheral unit is divided into independent measurement modules controlled by a master board. The number of slave modules to be installed depends on user needs and network needs.

Master CPU

The master CPU is a general-purpose Motorola 68000-based CPU board, largely configurable by means of programmable logic devices (PLD). The CPU has 512K bytes of RAM expandable to 2M bytes, 64K bytes of EPROM expandable to 512K bytes, 2K bytes of EEPROM, and a number of peripheral chips such as programmable input/output ports, serial communication controllers, and timers. The master CPU controls between one and four slave

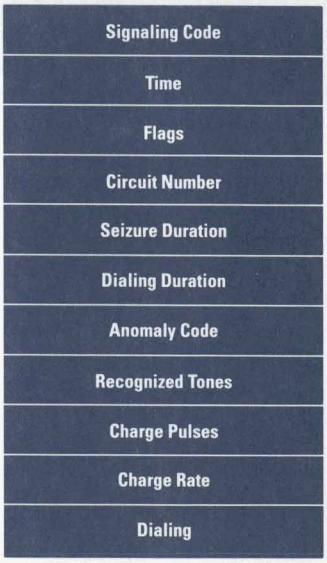


Fig. 2. Simplified call record format. The flags field shows the status of the seizure signal, answer signal, and answer complete flags.

modules through the system bus which includes data, address, interrupt, reset, monitoring, and control signals.

The master and slave CPUs are obtained from the same CPU board configured in different ways.

The interprocessor link is based on dual-port RAMs which are located on the slave CPU boards. These special components simplify the interface circuitry and allow rapid exchanges of large amounts of data with reduced interprocessor overhead.

The master CPU manages all the slave modules in the peripheral unit by:

- Bootstrapping
- Managing commands between the central unit and each slave module
- Managing messages between different slave modules (relay function)
- Uploading measurement data from each slave module to the central unit.

All communications with the central unit, whether via the communication board or the internal modem, are controlled by the master CPU. This allows the slave CPUs to focus on data collection and reduce the overhead for communication tasks.

In addition, the master CPU controls the portable operator interface, a simplified RS-232 interface which is used with a portable PC during installation operations or local diagnosis of the peripheral unit.

Slave Module

Each slave module consists of a CPU board named the slave CPU, a PCM interface board, and an optional digital signal processing board.

The CPU board controls the other two boards via a local bus similar to the one used by the master CPU, using a master-slave concept. A high-speed dedicated bus connects the PCM interface board and the digital signal processing board.

Depending on the user needs, each module can be remotely and independently configured. Moreover, the user can change the configuration at any time because all the application programs are located in RAM and bootstrapped during system startup.

Each slave module is able to:

- Monitor one 4-wire, 2-Mbit/s PCM link.
- Detect frame and multiframe PCM alarms.
- Analyze channel associated signaling (CAS) systems with pulse dialing.* This is performed simultaneously on all 30 channels in the stream.
- Analyze CAS systems with multifrequency dialing. The part of the signaling located in the signaling time slot is analyzed fully (100% coverage at the busiest times), whereas the in-channel multifrequency signaling is simultaneously inspected on up to 4 channels (corresponding to 97% coverage at the busiest times). This provides a good compromise between measurement performance and cost.
- Analyze common channel signaling (CCS) systems compatible with CCITT Signaling System #7 up to layer 2.*
- Detect service tones (such as ringing tone, busy tone and so on).

The main data collection functions of the module are provided by the PCM interface board under control of the slave CPU. When required, analysis of multifrequency signaling in the voice channels is performed by the digital signal processing board.

The PCM interface board consists of three basic parts: the PCM interface, the PCM stream memories, and the digital signal processing unit for signaling channel analysis. The PCM interface manages the two 2-Mbit/s PCM signals, which are in compliance with CEPT standards. The signals are decoded and synchronized before the frame octets are stored in memory.

*Application depends on the loaded software.

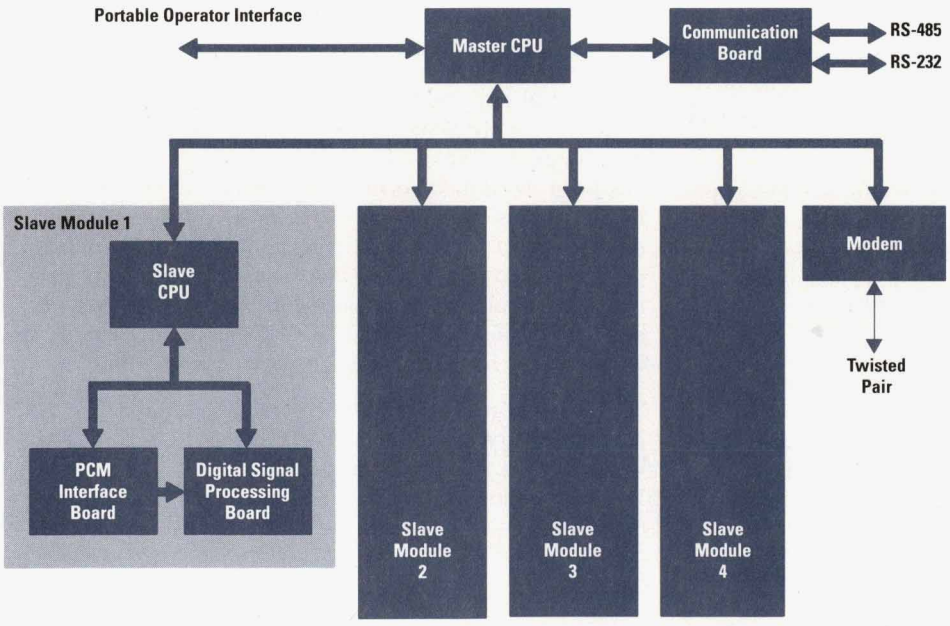


Fig. 3. Peripheral unit architecture.

Depending on the signaling system in use and the slave CPU settings, the 64-kbit/s data stream that is used for common channel signaling is extracted from each PCM stream.

The typical PCM alarms such as loss of incoming signal (LIS), loss of frame alignment (LFA), alarm indication signal (AIS), and others are detected, sampled, and stored in such a way that the events are recorded with the best accuracy and resolution in terms of their duration, for instance, 250 μs for AIS. The alarm circuitry is capable of revealing a loss of incoming signals with durations of only a few microseconds.

It is important to record these PCM alarms to be able to correlate service failures with the quality of the physical link.

Application-specific integrated circuit (ASIC) chips were developed to generate control and synchronization signals. These help increase reliability, reduce cost, and save board space.

The PCM stream memories consist of dual-port RAMs that are used for temporary storage of the PCM frame octets which appear every two milliseconds. When required, these 16-byte packets are transferred to the CPU and digital signal processing units for further processing. Special circuits are dedicated to controlling the data routing between the memories and the processing units. To reduce CPU loading, these circuits are responsible for downloading the PCM data packets to the selected digital signal processing unit and only need to be started by the CPU.

The digital signal processing unit, based on a Texas Instruments TMS320C10 processor working with a 20-MHz clock, processes the A-law-coded* bytes downloaded from the dual-port RAMs into FIFO chips. The use of FIFOs speeds up interface operations and simplifies the circuitry.

Use of digital processing allows complex and time-independent analysis, flexibility, and multiplexing. In addition, the application programs used by the PCM interface are loaded in RAM and can be changed at any time by the CPU depending on user needs.

The digital signal processing unit implements up to 16 service tone detectors (425-Hz tones) using reliable digital filtering techniques.

The PCM interface board offers a powerful self-test capability to detect hardware failures or malfunctions. For example, an on-board generated signal can be substituted for the external PCM signal; this signal simulates all the frame alarms and tests the specialized PCM circuits.

The slave CPU selects the operating mode of the module's boards and, after processing the signaling data, extracts information concerning telephone traffic. In the case of channel associated signaling, it analyzes the signaling time slot information (usually extracted from time slot 16), which is stored in the dual-port RAMs, and the digital signal processing results. For common-channel signaling, a two-channel serial communication controller analyzes the two data streams that come from the PCM interface board. The results of this analysis consist mainly of call and alarm records, which are stored in an internal database before being forwarded to the central unit via the master CPU.

If multifrequency signaling analysis is required, the slave module requires an optional digital signal processing board. This board contains three digital signal processing units that are identical to the single digital signal processing unit on the PCM interface board. In reality, only two of the units are used to realize four multifrequency receivers; the last one is a spare. The data to be processed is directly provided by the PCM interface board through a high-speed bus (20 Mbits/s peak).

A very simple hardware structure and digital processing based on a fast Fourier transform (FFT) algorithm ensure the detector's reliability. The efficiency of the detector is further increased by the use of Hamming windowing and shifting the analyzed spectrum to minimize the leakage errors resulting from the finite length of the time record.[1]

The peripheral unit's application software has been designed to be easy to adapt to the different signaling systems that are in use around the world. The different software measurement modules can be linked differently to achieve the desired results but they all use the same operating environment and services. This enables a rapid development process and ensures high software quality. Only the higher-level module used to decode the semantics of the signaling is developed specially for each signaling version. To simplify this process, this layer has been written using the Specification and Description Language (SDL), a very high-level language. This language, specified by the CCITT, has the advantage that it is well-known among signaling experts and allows reliable communication between specifiers and designers.

Finally, its physical dimensions make the peripheral unit suitable for simple installation in the 19-inch standard racks that are commonly found in central office premises.

The Central Unit

The central unit for the HP E3500A network monitoring system is an HP 9000 computer running the HP-UX operating system, version 7.0 or higher. The system software has been developed so that any member of the HP 9000 family (Series 300, 400, or 800) can be used, depending on the size of the network and the number of peripheral units required. The HP E3500A network monitoring system peripherals can be grouped in clusters (up to a maximum of 16 units on a single RS-485 bus) and each cluster is connected to the central unit using a serial line.

The central unit is responsible for:

- Managing all of the connected peripherals
- Collecting data from the peripherals
- Processing the collected data to obtain quality indexes
- Storing the quality indexes and all the elementary data

*During analog-to-digital conversion of a voice signal, A-law encoding logarithmically compresses the 12-bit value resulting from uniform quantization into an 8-bit word. The A law is recommended by the CCITT and is used in all countries except North America and Japan, where the μ law is used.

- Displaying the quality indexes (and, if required, elementary data) as required by the system operator.

The central unit software uses a relational database (Oracle RDBMS Version 6.0 for the current release) for all of its system configuration and elaboration needs. In the following descriptions this database is referred to as the "disk database", while "database" refers to a generic data base entity.

The central unit software can be divided into three main parts:

- The communications software manages the communications between the central unit and the peripheral units.
- The elaboration software provides interpretation and analysis of collected data.
- The user interface allows control of the system and displays network performance. The operator accesses the system through this OSF/Motif-style interface.[2]

In addition to these main parts, there is a supervisor function that manages system behavior and resources. The following descriptions focus on the elaboration software (Fig. 4), to show how the performance required for the HP E3500A network monitoring system was achieved.

Elaboration Software

In the HP E3500A network monitoring system the data sent from the peripherals to the central unit is always packed in the same format: the call record. A call record contains all the information about a single call attempted by a user of the monitored telephone network. It contains information such as: the start time of the call, dialed digits, duration, answer type, and so on. All of the call records coming from all of the peripherals are sent by the communication software to the ELAB_QUEUE message queue. The ELAB process takes each call record from ELAB_QUEUE and then:

- Finds the time slice, origin, and destination of the call.

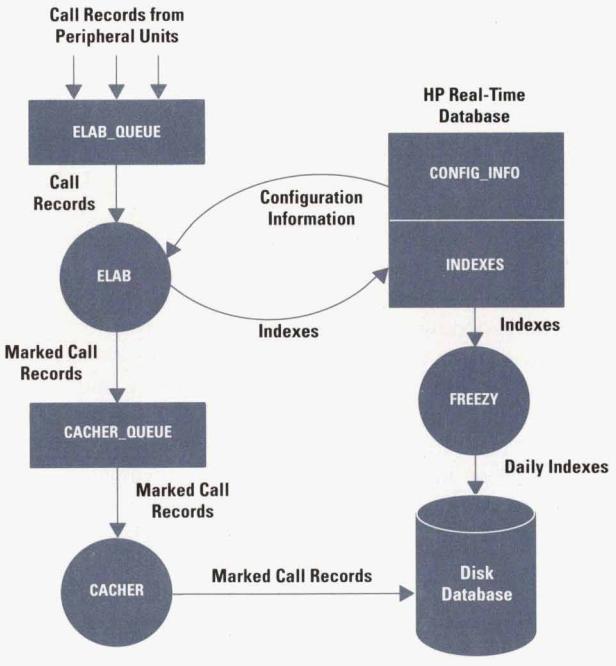


Fig. 4. Simplified block diagram of the elaboration software.

- Classifies the call (determines whether the call is successful or unsuccessful because of caller behavior, network congestion, destination busy, or any other reason).
- Updates the quality indexes for the current time slice, origin, and destination triple with the current call contribution.
- Stores each (classified) call record and the updated quality indexes in the disk database.

To perform the first three tasks, ELAB must read the required information (network topology, peripheral set description and other network-related information) from the disk database, and then write to the disk database for the last task. About ten disk database queries (one of which is sequential) are needed to complete these tasks for a single call record. At the same time, for a given number of peripherals, large amounts of data are being entered into ELAB_QUEUE via the communication software (that is, from the peripheral units). For example, a typical system consisting of 20 clusters with a total of 100 peripheral units must deal with about 300 call records per second. This is considered the target level of performance for a large HP 9000 Model 835-based system and is based on experimental data obtained from peripherals installed in the Italian telephone network. The size of a call record after the expansion performed by the communication software is 105 bytes. It is obvious that a consumer (ELAB) using the disk database cannot cope with a producer such as the HP E3500A network monitoring system peripheral units.

The solution adopted uses the HP Real-Time Database[3] (HP RTDB) as a cache memory before finally storing the data in the disk database. HP RTDB is a high-performance real-time database management system built on the HP-UX shared memory feature, which allows multiple processes to access the database concurrently.

The RTDB is initially created by the system manager during system configuration. Using some offline utilities that come with the system, the system manager can copy some of the tables required by ELAB from the disk database into the RTDB. These are then loaded into memory at system startup. When the system is running, ELAB can read the information it needs for call record elaboration from the RTDB (CONFIG_INFO in Fig. 4), and write the quality indexes into the RTDB (INDEXES in Fig. 4). At a rate of 300 call records per second or more, which would require at least 3000 database queries per second or more, the performance offered by the RTDB is far beyond current needs. By this method, the database access requirements of ELAB are no longer a bottleneck for the system.

HP RTDB is a memory-resident database, so there must be a process that definitively stores its data in the disk database. The process named FREEZY works in the background, continuously scanning the RTDB INDEXES table. When it finds an entry (day, time slice, origin, destination) in the table that has been marked as CLOSED by ELAB, it freezes it in the corresponding day table in the disk database as an entry consisting of time slice, origin, and destination. The algorithm used by ELAB to mark an entry applies a user-programmable time window (whose span is

given in units of time slices) which moves on with new incoming call records. When a time slice TS slides through the lower edge of the time window, ELAB marks all the entries belonging to TS (the old time slice) as CLOSED and discards any further incoming call records for that time slice (if any).

The only remaining problem is the call record storage process. Because of their total size—up to 1 Mbyte per day per peripheral unit, it is not possible to store all of the call records in the memory-resident RTDB, and ELAB cannot write them directly into the disk database. The solution that was implemented uses a new process (CACHER in Fig. 4) and a virtual-memory-based algorithm. ELAB now receives call records from ELAB_QUEUE, elaborates each of them as quickly as possible thanks to the HP RTDB, and simply sends them to the new CACHER_QUEUE message queue. CACHER is a very fast consumer. It continuously checks the CACHER_QUEUE message queue, reads any call records that it contains, and stores them in memory. Only when it is not busy (that is, when there is nothing to receive from CACHER_QUEUE), does it free the call records and put them in the disk database. The following listing is a nonformal pseudolanguage description of the CACHER_QUEUE algorithm:

```
flag = WAIT;
for (;;) {
    if (msgrcv(...,flag) == ENOMSG) {
      if (something in memory) free_and_store(...);
      else flag = WAIT;
    }
    else {
      malloc_in_memory(...);
      flag = NOWAIT;
    }
}
```

Since malloc() works with virtual memory, the run-time storage limit for CACHER is only limited by the disk size.

We performed a series of load tests on an HP 9000 Model 835 computer with 32M bytes of RAM. The following settings were made:

- The queue size was modified to the maximum allowed in HP-UX (64K bytes, equivalent to two seconds fill time at 300 call records per second).
- The semaphores and shared memory parameters were modified according to HP RTDB needs.
- The HP RTDB tables were LOCKED in memory to avoid disk swapping.
- The user was given LOCKRDONLY and RTPRIO privileges.
- CACHER was configured to store call records in memory in 100K-byte blocks and the following real-time priorities were given to the processes:
  - Protocol processes: time-sharing priority
  - Receiving processes: real-time priority 91
  - ELAB: real-time priority 90
  - CACHER: real-time priority 92.

The system was then loaded with programmable bursts of call records and the test results are shown in the following table.

HP E3500A Network Monitoring System Load Test Results on an HP 9000 Model 835

NRec	Pause	NLines	Call-Rec/s	CPU Load %	Disk Use %
15	500	2	55	30	70
15	500	4	110	55	60
15	500	10	230	70	35
15	250	10	285	85	20

NRec = call records/burst
Pause = time between two bursts in ms
NLines = number of serial lines in the system

Note that CPU load increases with throughput, while the disk use decreases because CACHER has less time left to store call records in the disk database. The queues never seemed to get full and the virtual memory usage can increase almost indefinitely, allowing it to follow peak throughput rates for a long time. From the table it can be seen that an HP 9000 Model 835 can manage a throughput of 250 to 300 call records per second, which could support a system containing 60 to 100 peripheral units.

Quality Indexes

Quality indexes must give an overview of network load and service quality. The procedure to compute them must deal with some basic requirements:

- It must be simple for the system administrator to define and/or modify index names and definitions.
- Indexes should give aggregate information.
- The computing procedure must be able to manage physical differences between network nodes (node main switching units can use different electrical models).
- It must be possible to examine computed values for possible network fault conditions.

  Computed values must match statistical significance criteria.

A description of how each of these requirements was fulfilled is given in the following sections.

Index Names and Evaluation Algorithms

The definition and modification of index names and the algorithms used to define them were made easy to perform and use by a programming approach. The user can describe indexes and their algorithms in a text file that is read and interpreted each time the system software is started. The programming language is simple to use yet powerful enough to allow arithmetic computations, variables, and flow control by means of IF-THEN-ELSE and WHILE constructs. The individual fields of call records are available for computation as predefined variables. Variables declared as indexes are automatically evaluated online for every call record received and subsequently stored in the system database. These elementary indexes consist of parameters that can be extracted directly from call records such as number of attempted calls, number of successful calls, and so on.

Aggregate Information

The need to get aggregate information—for instance, the mean duration of telephone calls between two nodes during a specific time period—was satisfied by introducing a second group of indexes. The algorithms to compute these compound indexes are called offline from the user interface and can use the values of the corresponding elementary indexes (recalled from the disk database for this purpose).

Physical Differences

Network pairs of origin link and destination are logically grouped in sets according to the signaling system adopted. The programming language takes this into account and provides the class construct to match each set with the related description. When ELAB updates a set of indexes, it uses the origin link and the destination node of the call record to index the proper class and the associated computing procedure.

Network Fault Detection

It is useful to have the system automatically detect possible fault conditions within the monitored network and focus operator attention on those faults. A simple way of doing it is to compare each evaluated index with a predetermined threshold value. Most indexes are defined so that faults are indicated by low values, so a possible fault will be linked to a value under the given threshold. However, there is not always a clear distinction between acceptable values and faulty ones. Therefore, it was decided to associate every index with two threshold values for the three following situations (lower to higher value):

- Alert/Possible Fault
- Warning
- OK.

The value of the fault test is returned with the index value after computation has been completed and is used in the display phase to highlight possible problems. The fault thresholds for an index can be redefined within each class.

Statistical Significance

Phone call parameters (number and duration) and aggregate indexes (mean duration, total number of calls in a time period, and so on) can be seen as random processes with given characteristics (unilateral exponential variables, binomial variables, Poisson processes, and the like). Thus the evaluated indexes can be used not only to see the state of the network at a certain time in the past, but also as a means of forecasting its future performance. This is, of course, one of the most meaningful uses of the HP E3500A network monitoring system in addition to network fault detection.

To be able to use an index value as an estimate (acceptable to a certain extent) for the characteristic of the random process it is associated with, the index must match the criteria of probability theory. To get an estimate with a given probability of maximum relative error (relative to the theoretical random process value), the system has to monitor a minimum number of calls.

This led to the introduction of two more thresholds for each index to cover the three situations:

- Statistically unacceptable value
- Low statistical significance
- Statistically significant value.

The intermediate level was introduced to take into account situations where a low number of samples deprives the index of its full statistical meaning but can nevertheless signal a network fault. As for the index threshold, the value of this test is returned with the index value after the computation and is used in the display phase combined with the fault test value. Like the fault thresholds, the statistical thresholds for an index can be redefined within each class.

Acknowledgments

The HP E3500A network monitoring system is the result of the common efforts of many engineers and managers who have worked on this project at the Necsy Telecommunications Operation in Padua, Italy. The knowledge, experience, and enthusiasm of these people contributed to the realization of a powerful and useful instrument for solving many telecommunication network problems. The authors wish to thank all the teams involved in this project. Special thanks go to the R&D team, in particular to: Massimo Mason, the project manager, who coordinated the different contributions, and to Antonello Gazzetta, Luca Vanuzzo, Alessandra De Nardi, Daniele Marazzato, Alberto Canella, Antonio Bovo, and Loredana Minello. Very special thanks go to Claudio Marangoni, Sergio Longo, and Ernesto Della Sala (HP Italy) whose contributions in the architectural design of the central unit software were essential.

Reference

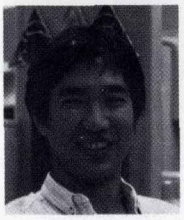
1. C. Offelli, et al, "A Real-Time Digital Detector for Bifrequential Coded Signaling," *Proceedings of MELECON 1991*, Ljubljana, May 1991.
2. *Hewlett-Packard Journal*, Vol.41, no. 3, June 1990, pp 6-35.
3. F. Fatehi, et al, "A Data Base for Real-Time Applications and Environments," *Hewlett-Packard Journal*, Vol. 40, no. 3, June 1989, pp. 6-17.

Authors

December 1991

6 HP Sockets

Mitchell J. Amino



Former HP summer intern Mitchell Amino joined HP's Advanced Manufacturing Systems Operation (AMSO) as a full-time software development engineer in 1985. After two years at AMSO, Mitch worked as a member of the marketing staff for HP's Technical Computing Operation and after that he joined the HP Sockets team. On the HP Sockets project Mitch was responsible for the user access routines, the data routing and storage module, and the administration modules. Mitch received his BS degree in computer science in 1983 from the University of Illinois at Urbana and an MS in computer science in 1985 from the University of Wisconsin at Madison. Mitch was born in Chicago, Illinois and he currently lives in San Jose, California. His leisure-time interests include volleyball, softball, outdoor activities, reading, and like many Chicagoans, Mitch is an avid Cubs fan.

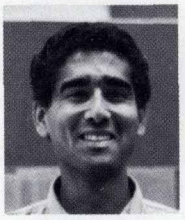
Irene S. Smith

Software design engineer Irene Smith, or "Skup" as she is known to her colleagues, was responsible for developing the HP Sockets gateway to non-HP platforms. She joined HP in 1987 at HP's Manufacturing Productivity Division. Irene received her BS degree (1981) in industrial engineering from the University of Wisconsin at Madison and an MS degree (1987) in electrical engineering from Carnegie-Mellon University. Before joining HP she was involved in developing robotic systems at Unimation/Westinghouse Corp. in Pittsburgh, Pennsylvania. Irene was born in Racine, Wisconsin and she now lives with her husband in Cupertino, California.

Mark Ikemoto

Former U.S. Army sergeant Mark Ikemoto joined HP's Advanced Manufacturing Systems Operation (AMSO) in 1985 after receiving his BS degree in computer science from San Francisco State University that same year. Mark also has a BA degree (1983) in psychology that he received from San Francisco State University. At AMSO Mike worked on the GMT400 project which involved automating factories for General Motors Corp. For the HP Sockets project, Mark was responsible for implementing the common data representation (CDR) protocol. Mark enjoys problem solving, especially the challenge of finding and fixing software bugs. His other interests include running, swimming, home improvement, writing stories, and good movies.

Alan C. Miranda



Software development engineer Alan Miranda began his engineering career as a civil engineer, receiving a B.Tech. in civil engineering in 1979 from the Indian Institute of Technology in Bombay, India and then an MS degree in civil engineering (major in structures) in 1980 from the University of Michigan at Ann Arbor, Michigan. In line with his civil engineering education, Alan worked for Quadrex Corporation as a pipe support engineer designing supports for nuclear power plants. Alan joined HP's Advanced Manufacturing Systems Operation (AMSO) in 1986 while working on an MS degree in computer engineering at San Jose State University in San Jose, California. He completed his degree in 1987. At AMSO Alan worked as a software development engineer on the GMT400 (truck and bus) project. He later joined HP's Technical Computer Operation and worked as a product marketing manager responsible for HP 9000 Series 800 networking. On the HP Sockets project, Alan was responsible for the development of the network-wide HP Sockets startup and shutdown functions. Alan was born in Ahmedabad, India and now lives in San Jose, California. His recreational interests include running, weightlifting, tennis, and bowling. He plans to try camping and hiking very soon.

Kathleen A. Fulton

Development of the data manipulation modules was among the tasks software development engineer Kathleen A. Fulton worked on for the HP Sockets project. Kathy joined HP's Engineering Productivity Division in 1984 and she helped develop the equipment control portion of HP's Semiconductor Productivity Network. She was also responsible for implementing portions of the HP Device Interface System (HP DIS). Before joining HP, Kathy was a semiconductor fabrication engineer at Burroughs Corp. and a software development engineer at Trilogy Corp. and Amdahl Corp. A member of the IEEE, she received a BA degree in computer science in 1975 from the University of California at San Diego. Born in Reno, Nevada, Kathy is married and lives in Cupertino, California. Her interests include travel, reading science fiction, and nice long walks.

Cynthia Givens

The HP Sockets incoming and outgoing network managers were among the modules that software development engineer Cynthia Givens worked on for the HP Sockets project. Before working on the HP Sockets project, Cynthia worked on the HP Real-Time Database product, and she was also a coauthor of an HP Journal article about the product. Cynthia joined HP in 1983 after receiving a BA degree in computer science from the University of Texas at Austin. Her first projects at HP included the MMC/1000 manufacturing application and the AGP/DGL graphics packages. Born in Durango, Colorado, she's married and lives in Santa Clara, California. She enjoys outdoor activities such as hiking, biking, and running.

Scott A. Gulland

For the HP Sockets product Scott Gulland was the lead design engineer. He was responsible for the design and development of the data definition and data manipulation compilers, the error logging modules, and the memory management libraries for the request manager and the run-time configuration table. Scott is now a development engineer on the telecom team at HP's Advanced Manufacturing Systems Operation. In the past he has worked on DataCAP/1000, the HP 1000 A-Series Link, the Quality Decision Management/1000 package, and on the GMT400 project as lead engineer. The GMT400 project developed workcell controllers for General Motors' truck and bus manufacturing plants. Scott joined HP's Data Systems Division in 1980 after receiving a BS degree in computer science from California Polytechnic State University at San Luis Obispo. He was born in San Antonio, Texas and currently lives in San Jose, California with his wife and two children. Scott's outside-of-work interests include camping, backpacking, and duplicate bridge.

24 Rigorous Software Engineering

Tony W. Rush

After receiving a BSc degree (1985) in mathematics from the University of Manchester in Manchester, England, Tony Rush joined HP Laboratories in Bristol, England. Tony has been a member and leader of several HP-SL transfer projects, in which the formal methods technology is transferred to other HP divisions. He is also the codeveloper of the HP-SL training course. Tony is a member of the British Computer Society and is interested in industrializing the

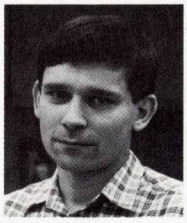
use of formal methods. He was born in Lisburn, Northern Ireland. He is married and his current interest outside of work is to help take care of and enjoy his new son.

Stephen P. Bear

Managing research efforts to demonstrate that rigorous software engineering technologies can improve software development in HP is the primary duty of project manager Stephen Bear. Steve joined HP Laboratories in Bristol, England in 1988. His initial assignments involved research into the specification and design of object-oriented software. He has a BA degree (1976) in mathematics and a PhD in mathematics (1980) from the University of Lancaster in Lancaster, England. Before joining HP Steve was at the United Kingdom Central Electricity Generating Board working on software development, computer system management, and software engineering methods. His professional interests include the use of formal specification to improve the efficiency and quality of software development. He has also published five papers on formal specification. Steve was born in London, England and currently lives in Bristol, England with his wife and two children.

32 Electronic Mail System

Patrick C. Goldsack



A member of the technical staff in the applied methods group at HP Laboratories in Bristol, England, Patrick Goldsack was one of the designers of the HP-SL notation. He has also worked on course material, documentation, and support tools for HP-SL. Patrick received a BA degree in mathematics from Oxford University in 1978 and an MSc degree in 1982 in electrical engineering from Aston University in Birmingham, England. He joined HP Laboratories in Bristol in 1987. Before joining HP, Patrick worked as a software engineer on advanced software methods at STC Research, and before that as an engineer at GEC Telecommunications designing automatic test equipment for telephone systems. Patrick was born in Birmingham, England and currently resides in Bristol with his wife and two children. He is the current chairman of the HP Wine Tasting Society, and when he's not sampling wine he enjoys bridge and photography.

Tony W. Rush

Author's biography appears elsewhere in this section.

40 Real-Time Behavior

Paul D. Harry

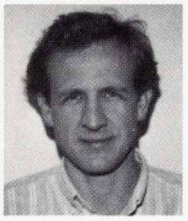
Providing consultation, giving courses, and working in collaboration with HP divisions using rigorous software engineering techniques have been Paul Harry's main contributions to the HP-SL project. Paul came to HP Laboratories in Bristol in 1988 and is currently a member of the technical staff in the applied methods group of this entity. Paul has a BSc degree (1984) in mathematics and physics from the University of Bristol and an MSc degree (1988) in computer science from the University of Manchester. Before joining HP Paul worked at System Designers Ltd in Portland, England designing and implementing command and control systems. Paul is very interested in seeing that formal methods are included in the software development process.

Tony W. Rush

Author's biography appears elsewhere in this section.

46 HP-SL in Product Development

B. Robert Ladeau



Development engineer Robert Ladeau came to HP's Waltham Division in 1985 and now works in the patient monitor/cardiac care systems business unit of that division. Robert holds an AB degree (1977) in chemistry from Middlebury College in Middlebury, Vermont and an MS degree (1988) in computer science from Boston University. He is very familiar with ST segment analysis because he worked on adding ST segment analysis and other ECG related functionality to the HP 7834C patient monitor terminal, and he is currently working on enhancing the ST functionality in another HP patient monitor. Formal specification, requirements analysis, and user interface design are Roberts's professional interests. He currently resides in Andover, Massachusetts with his wife and two children. His outside-of-work activities include running and tennis.

Curtis W. Freeman

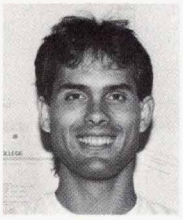
A software development engineer in the patient monitor/cadiac care business unit of HP's Waltham Division, Curtis Freeman works on ST segment analysis for bedside monitors. Curt joined the Waltham Division in 1985. Before joining HP he worked at Eastman Kodak Company as a software engineer for a materials management system and as a manufacturing engineer for the photochemicals division of that company. Curt has a BS degree (1980) in chemical engineering and an MS degree (1985) in information systems from Northeastern University in Boston, Massachusetts. He was born in Norfolk, Massachusetts and he currently lives in Windham, New Hampshire with his wife and three children. He is a church youth group leader and he likes to indulge in skiing, soccer, and home brewing.

51 HP-SL and Structured Design

Judith L. Cyrus

Software development engineer Judith Cyrus works at HP's McMinnville Division in the clinical business unit. Judi began her HP career in Germany at HP's Böblingen General Systems Division in 1986, providing technical support for HP Business Report Writer/V. She also worked as a software development engineer at HP's Böblingen Medical Division. Judi has a BS degree (1986) in computer science from the University of Montana and is a member of the IEEE. She was born in Grand Junction, Colorado and currently lives in Newberg, Oregon. Judi is married and has three daughters whose first names all begin with the letter J. Landscaping, animal care, horseback riding, reading, and crafts are among her hobbies and interests.

J. Daren Bledsoe



Daren Bledsoe is a project manager for Holter patient monitor units at HP's McMinnville Division. He came to HP in 1983 after receiving a BSEE from Washington State University in Pullman, Washington. He has worked as a hardware design engineer and as a production engineer in the development of the HP 43400A, 43401A, 43400B, 43402A, and 43405A Holter monitor patient units. Daren was born in Livermore, California and currently lives in McMinnville, Oregon. He is married and has two sons. His outside-of-work activities include flying and running.

Paul D. Harry

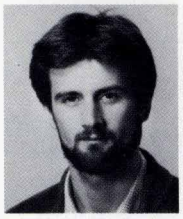
Author's biography appears elsewhere in this section.

59 Network Monitoring System

Marco Silvestri

Marco Silvestri is a software engineer at the HP Necsy Telecommunications Operation in Padua, Italy. He was the user interface developer for the HP E3510A software for the HP E3500A network monitoring system. A native of Padua, he received his electronic engineering degree in 1988 from Padua University and served in the military transmission service for 15 months as an auxiliary official. He joined Necsy in 1989.

Antonio Posenato

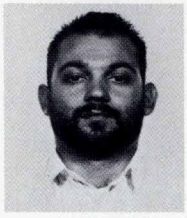


Antonio Posenato is a senior R&D engineer at the HP Necsy Telecommunications Operation in Padua, Italy. A 1984 electronics graduate of Padua University, he joined Necsy in 1986. He was responsible for the firmware of the HP E3500A network monitoring system, and also did hardware design of the digital boards. He's now system manager for the E3500A peripheral units. A native of Verona, he served in the military transmission service for 15 months as an auxiliary official. His professional interests are data transmission and ISDN, and his outside interests include sports (soccer, tennis, skiing, running) and oil and watercolor painting.

Giuseppe Mazzucato

R&D project manager Beppo Mazzucato was a system engineer for the HP E3500A network monitoring system. He received his electronic engineering degree in 1987 from Padua University and joined the HP Necsy Telecommunications Operation the same year. He's also done software development and system specification. Beppo was born in Padua, Italy and has done military service in the Italian infantry. He is married, sings in a chorus, and plays tennis and volleyball.

Nicola De Bello



Nick De Bello is a 1987 electronics graduate of Padua University and a system engineer at the HP Necsy Telecommunications Operation. He joined Necsy in 1987 and was responsible for the HP E3510A software for the HP E3500A network monitoring system. His professional interests include software engineering, UNIX, X, and OSF/Motif programming, and real-time systems. Born in Udine, Italy, he served in the Carabinieri for a year and is a volunteer in the Padua First Aid Ambulance Service. He is married, plays American football as fullback for the Padua Saints, and enjoys reading, playing guitar, painting, and role-playing.

HEWLETT-PACKARD
JOURNAL
INDEX

Volume 42 January 1991 through December 1991

Hewlett-Packard Company, P.O. Box 51827, Palo Alto, California 94303-0724 U.S.A.
Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan
Hewlett Packard (Canada) Ltd. 6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

Part 1: Chronological Index

February 1991

High-Speed Lightwave Component Analysis to 20 GHz, *Roger W. Wong, Paul R. Herndday, and Daniel R. Harkins*

Design of a 20-GHz Lightwave Component Analyzer, *Paul R. Herndday, Geraldine A. Conrad, Michael G. Hart, and Rollin F. Rawson*
Measurement Capabilities of the HP 8703A Lightwave Component Analyzer and the HP 71400C Lightwave Signal Analyzer

20-GHz Lightwave Test Set and Accessories, *Joel P. Dunsmore and John V. Vallelunga*

Accuracy Considerations and Error Correction Techniques for 20-GHz Lightwave Component Analysis, *Daniel R. Harkins and Michael A. Heinzelman*

Development of an Optical Modulator for a High-Speed Lightwave Component Analyzer, *Roger L. Jungerman and David J. McQuate*

High-Performance Optical Isolator for Lightwave Systems, *Kok-Wai Chang, Siegmar Schmidt, Wayne V. Sorin, Jimmie L. Yarnell, Harry Chou, and Steven A. Newton*

A Broadband, General-Purpose Instrumentation Lightwave Converter, *Christopher M. Miller and Roberto A. Collins*

A Lightwave Multimeter for Basic Fiber Optic Measurements, *Bernd Maisenbacher and Wolfgang Reichert*

Design of a Series of High-Performance Lightwave Power Sensor Modules, *Jochen Rivoir, Horst Schweikardt, and Emmerich Müller*

Calibration of Fiber Optic Power Meters, *Christian Hentschel*

Semiconductor Laser Sources with Superior Stability for Optical Loss Measurements, *Frank A. Maier*

Lightwave Multimeter Firmware Design, *Wilfried Pless, Michael Pott, and Robert Jahn*

A Visual User Interface for the HP-UX and Domain Operating Systems, *Mark A. Champine*
Open Dialogue
HP Visual User Interface, Version 2.0

April 1991

A Family of High-Performance Synthesized Sweepers, *Roger P. Oblad, John R. Regazzi, and James E. Bossaller*
Designing for Low Cost of Ownership
Strife Testing the Alphanumeric Display
Front Panel Designed for Manufacturability

Built-in Synthesized Sweeper Self-Test and Adjustments, *Michael J. Seibel*
Automatic Frequency Span Calibration
Accessing a Power Meter for Calibration

A High-Performance Sweeper Output Power Leveling System, *Glen M. Baker, Mark N. Davidson, and Lance E. Haag*
Mismatch Error Calculation for Relative Power Measurements with Changing Source Match

A 0.01-to-40-GHz Switched Frequency Doubler, *James R. Zellers*

A High-Speed Microwave Pulse Modulator, *Mary K. Koenig*

New technology in Synthesized Sweeper Microcircuits, *Richard S. Bischof, Ronald C. Blanc, and Patrick B. Harper*
Modular Microwave Breadboard System
Quasi-Elliptic Low-Pass Filters

DC-to-50-GHz Programmable Step Attenuators, *David R. Veteran*

50-to-110-GHz High-Performance Millimeter-Wave Source Modules, *Mohamed M. Sayed and Giovonnae F. Anderson*
The Use of the HP Microwave Design system in the W-Band Tripler Design
The use of HP ME 10/30 in the W-Band Tripler Design
Flatness Correction
High-Power W-Band Source Module

An Instrument for Testing North American Digital Cellular Radios, *David M. Hoover*
HP 11846A Filtering Technique

Measuring the Modulation Accuracy of π/4 DQPSK Signals for Digital Cellular Transmitters, *Raymond A. Birgenheier*

A Test Verification Tool for C and C++ Programs, *David L. Neuder*

June 1991

HP 48SX Scientific Expandable Calculator: Innovation and Evolution, *William C. Wickes and Charles M. Patton*

The HP 48SX Interfaces and Applications, *Ted W. Beers, Diana K. Byrne, Gabe L. Eisenstein, Robert W. Jones, and Patrick J. Megowan*

HP Solve Equation Library Application Card, *Eric L. Vogel*

Hardware Design of the HP 48SX Scientific Expandable Calculator, *Mark A. Smith, Lester S. Moore, Preston D. Brown, James P. Dickie, David L. Smith, Thomas B. Lindberg, and M. Jack Muranami*
Industrial Design of the HP 48SX Calculator
HP 48SX Custom Integrated Circuit
Mechanical Design of the HP 48SX Memory Card and Memory Card Connector

The HP 48SX Calculator Input/Output System, *Steven L. Harper and Robert S. Worsley*

Manufacturing the HP48SX Calculator, *Richard W. Riper*

A 10-Hz-to-150-MHz Spectrum Analyzer with a Digital IF Section, *Kirsten C. Carlson, James H. Cauthorn, Timothy L. Hillstrom, Roy L. Mason, Joseph F. Tarantino, Jay M. Wardle, and Eric J. Wicklund*
Spectrum Analyzer Self-Calibration
Adaptive Data Acquisition
Help System with Hypertext
User Interface Compiler

Easy-to-Use Performance Tools with a Consistent User Interface across HP Operating Systems, *Rex A. Backman*
Design Prototyping for HP GlancePlus
The Performance Tool Quadrant

Improving the Product Development Process, *Spencer B. Graves, William P. Carmichael, Douglas Daetz, and Edith Wilson*

DSEE: A Software Configuration Management Tool, *David C. Lubkin*

A Mechanism to Support Parallel Development via RCS, *John W. Goodnow*

Building and Managing an Integrated Project Support Environment, *Ronald F. Richardson*

October 1991

Introduction to the HP Component Monitoring System, *Christoph Westerteicher*
Medical Expectations of Today's Patient Monitors

Component Monitoring System Hardware Architecture, *Christoph Westerteicher and Werner E. Heim*

Component Monitoring System Software Architecture, *Martin Reiche*
Component Monitoring System Software
Component Monitoring System Software Development Environment

Component Monitoring System Parameter Module Interface, *Winfried Kaiser*

Measuring the ECG Signal with a Mixed Analog-Digital Application-Specific IC, *Wolfgang Grossbach*

A Very Small Noninvasive Blood Pressure Measurement Device, *Rainer Rometsch*

A Patient Monitor Two-Channel Stripchart Recorder, *Leslie Bank*

Patient Monitor Human Interface Design, *Gerhard Tivig and Wilhelm Meier*

Globalization Tools and Processes in the HP Component Monitoring System, *Gerhard Tivig*

The Physiological Calculation Application in the HP Component Monitoring System, *Steven J. Weisner and Paul Johnson*

Mechanical Implementation of the HP Component Monitoring System, *by Karl Daumüller and Erwin Flachsländer*

An Automated Test Environment for a Medical Patient Monitoring System, *Dieter Göring*

Production and Final Test of the HP Component Monitoring System, *Otto Schuster and Joachim Weller*

Calculating the Real Cost of Software Defects, *William T. Ward*

A Case Study of Code Inspections, *Frank W. Blakely and Mark E. Boles*

The HP Vectra 486 Personal Computer, *Larry Shintaku*
The HP Vectra 486 EISA SCSI Subsystem
The HP Vectra 486/33T
The EISA Connector, *Michael B. Raynham and Douglas M. Thom*
EISA Configuration Software

The HP Vectra 486 Memory Controller, *Marilyn J. Lang and Gary W. Lum*

The HP Vectra 486 Basic I/O System, *Viswanathan S. Narayanan, Thomas Tom, Irvin R. Jones, Jr., Philip Garcia, and Christophe Grosthor*

Performance Analysis of Personal Computer Workstations, *David W. Blevins, Christopher A. Bartholomew, and John D. Graf*

December 1991

HP Software Integration Sockets: A Tool for Linking Islands of Automation, *Mitchell J. Amino, Cynthia Givens, Mark Ikemoto, Alan C. Miranda, Scott A. Gulland, Kathleen A. Fulton, and Irene S. Smith*
Configuration Files
Performance in the HP Sockets Domain
HP Sockets Gateway

Rigorous Software Engineering: A Method for Preventing Software Defects, *Stephen P. Bear and Tony W. Rush*

Specifying an Electronic Mail System with HP-SL, *Patrick G. Goldsack and Tony W. Rush*
Specification of State in HP–SL

Specifying Real-Time Behavior in HP-SL, *Paul D. Harry and Tony W. Rush*
History Specifications

Using Formal Specification for Product Development, *B. Robert Ladeau and Curtis W. Freeman*

Formal Specification and Structured Design in Software Development, *Judith L. Cyrus, J. Daren Bledsoe and Paul D. Harry*

Telecommunications Network Monitoring System, *Nicola De Bello, Giuseppe Mazzucato, Antonio Posenato, and Marco Silvestri*

Part 2: Subject Index

Subject **Page/Month**

A

Absorption bands . . . . . 50/Apr.
Abstract data type . . . . . 32/Dec.
Access routines, HP Sockets . . . . . 9/Dec.
Adapters, HP Sockets . . . . . 9/Dec.
Adaptive data acquisition 51/June
Administration node, HP Sockets . . . . . 15/Dec.
Alarm monitor . . . . . 40/Dec.
Alarms . . . . . 16, 32/Oct.
ALC . . . . . 30/Feb.
ALC, sweeper . . . . . 24/Apr.
Amplifier doubler, R-band . . . . . 58/Apr.
Amplifier doubler, V-band . . . . . 56/Apr.
Amplifier tripler, W-band . . . . . 54/Apr.
Analyzer, lightwave component . . . 13/Feb.
Application management, HP 48SX . 9/June
Array of choices . . . . . 34/Oct.
ASICs . . . . . 11, 22, 23/Oct.
ASN.1 . . . . . 21/Dec.
Atmospheric windows . . . . . 50/Apr.
Attenuators, programmable, step . 47/Apr.
AUTOMAN keypusher . . . . . 50/Oct.
AUTOTEST . . . . . 49/Oct.

B

Basic Encoding Rules (BER) . . . . . 21/Dec.
BIOMON (backplane I/O activity monitor) . . . . . 94/Oct.
BIOS (Basic I/O System), Vectra 486 . . . . . 83/Oct.
Birefringent crystals . . . . . 46/Feb.
Bismuth-substituted YIG films . . . . 46/Feb.
Blood pressure . . . . . 6, 25/Oct.
Bondless microcircuits . . . . . 38/Apr.
Bottom case assembly, HP 48SX . . 30/June
Branch analysis . . . . . 83/Apr.
Branches, DSEE 78/June
Breadboard system, microwave . . . 41/Apr.
Break-even time (BET) 71/June
Build management 85/June
Burst-mode read, Vectra 486 . . . . . 81/Oct.
Bus master . . . . . 73/Oct.

C

C++, HP Branch Validator . . . . . 91/Apr.
Cache memory, Vectra 486 . . . . . 78/Oct.
Cache simulator . . . . . 95/Oct.
Calculation evaluator . . . . . 42/Oct.
Calculator, scientific expandable . . . 6/June
Calibration, lightwave analyzer . . . . . 20, 34/Feb.
Calibration, optical power meter . . 70/Feb.
Calibration, self, spectrum analyzer 47/June
Calibration, sweeper . . . 19, 21, 22, 26/Apr.
Call record . . . . . 60/Dec.
Cardiac work (LCW/RCW) . . . . . 41/Oct.
Cellular system (NADMCS) . . . . . 65/Apr.
Central plane . . . . . 7, 12/Oct.
Central unit, network monitor . . . . 62/Dec.
Check out and check in 87/June
Chromatic dispersion . . . . . 9/Feb.
Circular interpolation . . . . . 32/Feb.
Client server model . . . . . 20/Dec.
Code inspections . . . . . 58/Oct.
Common data representation . . . . . 20/Dec.
Communication model . . . . . 16/Oct.
Communication protocol . . . . . 20/Oct.
Compiler (mtc) . . . . . 15/Oct.
Compiler, NLS text . . . . . 39/Oct.
Compiler, user interface 57/June
Component Monitoring System . . . . 6/Oct.
Computer module . . . . . 7/Oct.
Computing environment 90/June
Computing support model 94/June
Configuration, automated . . . . . 17/Oct.
Configuration files . . . . . 13/Dec.
Configuration threads, DSEE 80/June
Converter, lightwave . . . . . 51/Feb.
Cost of ownership, sweeper . . . . . 10/Apr.
Coupler detectors, V and W bands . 53/Apr.
CPU cards . . . . . 7/Oct.
Crossover frequency, amplifier . . . 52/Feb.
Customization, HP 48SX 15/June

D

Daemon, HP Sockets . . . . . 14/Dec.
Database, real-time . . . . . 63/Dec.
Data definition language (DDL) . . . 21/Dec.
Data flow diagrams . . . . . 52/Dec.
Data management package . . . . . 41/Oct.
Data manipulation . . . . . 8, 20/Dec.
Data manipulation language (DML) . . . . . 22/Dec.
Data transceivers, Vectra 486 . . . . . 80/Oct.
Dc-to-dc converter . . . . . 7/Oct.
Decision points . . . . . 68/Apr.
Design for manufacturability . . . . . 15/Apr.
Design prototyping, HP GlancePlus 69/June
Digital cellular radios . . . . . 65/Apr.
Digital cellular transmitters . . . . . 73/Apr.
Digital IF, spectrum analyzer . . 44, 49/June
Digital modulation . . . . . 66/Apr.
Digital value placement . . . . . 34/Oct.
Directory links 86/June
Diskless workstation 84/June
Display front assembly . . . . . 45/Oct.
Displays, patient monitor . . . . . 7, 12/Oct.
Domain O/S 77/June
Double-width parameter module . . 27/Oct.
Doubler, 40-GHz switched . . . . . 31/Apr.
DQPSK modulation . . . . . 67/Apr.
Dual laser source . . . . . 76/Feb.
Dual-wavelength capability . . . . . 16/Feb.
dword . . . . . 78/Oct.

E

ECG waves . . . . . 46/Dec.
Edgeline attenuator design . . . . . 47/Apr.
EISA configuration software . . 75, 84/Oct.
EISA connector . . . . . 73, 75/Oct.
EISA consortium . . . . . 74/Oct.
EISA (Extended Industry Standard Architecture) . . . . . 69, 73/Oct.
EISA initialization . . . . . 84/Oct.
Elaboration, network data . . . . . 63/Dec.
Electrocardiogram (ECG) . . . . . 6, 21/Oct., 40/Dec.
Electronic mail system, HP-SL . . . . 32/Dec.
Equation library card 22/June
EquationWriter application 15/June
Error correction, lightwave . . . 21, 34/Feb.
Error vector magnitude . . . . . 73/Apr.
Event types . . . . . 41/Dec.
Exception-based reporting 66/June
Execution model . . . . . 16/Oct.
Execution trees . . . . . 16/Oct.
Extension, Command Set 80 . . . . . 81/Feb.
Extinction ratio . . . . . 44/Feb.

F

Fabry-Perot sensors . . . . . 11/Feb.
Feedforward ALC . . . . . 25/Apr.
Filtering, HP 11846A . . . . . 71/Apr.
Filters, quasi-elliptic . . . . . 42,44/Apr.
FIR filter, HP 11846A . . . . . 69/Apr.
Firmware, lightwave multimeter . . 77/Feb.
Firmware, patient monitor . . . . . 14/Oct.
Firmware, spectrum analyzer 57/June
Flatness correction . . . . . 59/Apr.
Formal specification . . . . . 46, 51/Dec.
Formal specification language . . . . 26/Dec.

Function cards ..... 7/Oct.
Functions, HP-SL ..... 29/Dec.

G

Gate array, power sensor ..... 68/Feb.
Globalization ..... 37/Oct.
Graded-index lens ..... 63/Feb.
Graphics, HP 48SX 17/June

H

Hardware architecture, patient monitor ..... 10/Oct.
Hardware design, HP 48SX 25/June
Harmonic analysis ..... 60/Apr.
Help, context sensitive ..... 32/Oct.
Heterogeneous configuration management 81/June
Heterogeneous environment, HP Sockets ..... 8/Dec.
Hexpander ..... 39/Oct.
Hifsim ..... 30/Oct.
High-resolution ADC ..... 67/Feb.
History specifications ..... 43/Dec.
History types, HP-SL ..... 41/Dec.
HP Specification Language (HP-SL) ..... 27/Dec.
HP StarLan 10 ..... 91/June
HP Sockets management daemon (SMD) ..... 14/Dec.
HP VUE 2.0 ..... 97/Feb.
Human interface, patient monitor . 29/Oct.
Hypertext help system 53/June

I

IF amplifier ..... 42/Apr.
Indexes, network quality ..... 64/Dec.
In-process project retrospective reviews 73/June
Input/output system, HP48SX 35/June
Integral contacts ..... 38/Apr.
Integration, test system ..... 53/Oct.
Intel486 ..... 69, 78/Oct.
Interface, parameter module .. 17, 19/Oct.
Interfaces, HP 48SX 36, 37/June
ISA (Industry Standard Architecture ..... 73/Oct.
Isolator, optical ..... 16/Feb.

J

K

Kermit protocol 36, 39/June
Keypad, patient monitor ..... 34/Oct.

L

Laser FM response ..... 9/Feb.
Laser source ..... 59, 73/Feb.
Lightwave component analysis, 20-GHz ..... 6/Feb.
Lightwave multimeter ..... 58/Feb.
Lightwave receiver ..... 19, 30/Feb.
Lightwave source ..... 7, 15, 29, 73/Feb.
Lightwave test set ..... 15, 23/Feb.
Lithium niobate ..... 42/Feb.
LO amplifier ..... 40/Apr.
LO feedthrough nulling 47/June
Local oscillator, spectrum analyzer 50/June
Localization ..... 37/Oct.
Loss measurements, optical ..... 60/Feb.
Low-band microcircuit ..... 36, 39/Apr.

M

Mach-Zender interferometer ..... 42/Feb.
Magnetooptic isolator ..... 46/Feb.
Mainline 77, 84/June
Manufacturing, HP 48SX 40/June
Manufacturing, patient monitor ... 52/Oct.
Maps, HP-SL ..... 28/Dec.
Master CPU ..... 60/Dec.
Material flow, vertical ..... 52/Oct.
Measurement interface model 68/June
Mechanical design, patient monitor ..... 44/Oct.
Memory architecture, Vectra 486 .. 79/Oct.
Memory card and connector 32/June
Memory controller, Vectra 486 .... 78/Oct.
Memory initialization, Vectra 486 ..... 90/Oct.
Memory subsystem simulator .... 95/Oct.
Message classes ..... 16/Oct.
Message passing bus ..... 8, 11/Oct.
Metrics database ..... 55/Oct.
Microcircuit design techniques ... 36/Apr.
Micro-DIN ..... 87/Oct.
Microwave design system ..... 53/Apr.
Mismatch error ..... 28/Apr.
Mixer, triple balanced ..... 40/Apr.
Modsplitter, microcircuit ..... 36, 43/Apr.
Modulator, optical ..... 18, 41/Feb.
Modulator, pulse ..... 34/Apr.
Module rack ..... 7/Oct.
Module specifications ..... 53/Dec.
Module tables ..... 17/Oct.
Monitor configuration table 17/Oct
Monitor, patient ..... 6/Oct.
Monitor, telephone network ..... 59/Dec.
Multimeter, lightwave ..... 58/Feb.
Multiple equation solver 23/June
Multiplying DAC ..... 66/Feb.
Multiprocessor system ..... 10/Oct.

N

Network interface ..... 12/Dec.
Network monitoring system, telephone ..... 59/Dec.
North American dual-mode I-Q diagrams ..... 66/Apr.
NLS database ..... 38/Oct.
NLS tools ..... 39/Oct.
Nulling, LO feedthrough 47/June

O

Object types, RPL 8/June
Open Dialogue, HP VUE ..... 93/Feb.
Optical launch measurement ..... 10/Feb.
Optical power measurements .... 58/Feb.
Optical reflection and transmission measurements ..... 9, 25/Feb.
OSF/Motif, HP VUE ..... 90/Feb.

P

π/4 DQPSK modulation ..... 65/Apr.
Pace pulse detection circuit ..... 23/Oct.
Parameter modules ..... 7, 19, 47/Oct.
Parameterized outer loop 13/June
Patient monitoring system 6/Oct
Patient simulators ..... 50/Oct.
Performance, HP Sockets ..... 16/Dec.
Performance, software 65/June
Performance tool quadrant 70/June
Performance, Vectra 486 ..... 92/Oct.
Peripheral units ..... 60/Dec.
Personal computer, Vectra 486 .... 69/Oct.
Physiological calculations ..... 40/Oct.
Plotting, HP 48SX 17/June
Plug-in management, HP 48SX 9/June
PMON (process activity monitor) ..... 92/Oct.
Poiseuille's law ..... 41/Oct.
Polarization controller ..... 17/Feb.
Post-introduction product reviews 72/June
Power leveling, sweeper ..... 24/Apr.
Power measurements, optical .... 58/Feb.
Power sensors, optical ..... 63/Feb.
Precision, HP-SL ..... 48/Dec.
Preprocessors, HP Branch Validator ..... 84/Apr.
Printed circuit assembly, HP 48SX . 29/June
Printhead control ..... 28/Oct.
Process specification, HP-SL ..... 54/Dec.
Product development process 71/June
Programmable-gain amplifier ..... 66/Feb.
Project management, DSEE 79/June
Pulmonary vascular resistance (PVR) ..... 41/Oct.
Pulse oximeter (SaO_2) ..... 6/Oct.
Pump assembly ..... 25/Oct.

Q

QPSK modulation ..... 67/Apr.
Quadratic gradient constant ..... 64/Feb.
Quality function deployment (QFD) 74/June
Quality, manufacturing process ... 54/Oct.
Quasi-elliptic low-pass filters .. 42, 44/Apr.

R

R2 signaling system ..... 59/Dec.
Rack interface controller ..... 20/Oct.
RCS (revision control system) 84/June
Real-time specifications, HP-SL ... 40/Dec.
Receiver, lightwave ..... 19, 30/Feb.
Receiver, spectrum analyzer 45/June
Recorder, stripchart 26/Oct
Reference/trigger section, spectrum analyzer 53/June
Refinement, HP-SL ..... 38/Dec.
Reflection measurements, lightwave ..... 11/Feb.
Remapping, Vectra 486 ..... 89/Oct.
Report generator, HP Branch Validator ..... 88/Apr.
Resolution bandwidth filters, sweeping 55/June
Resting display ..... 33/Oct.
RF deck, sweeper ..... 8/Apr.
RF test set ..... 13/Feb.
Rigorous software engineering ... 25/Dec.
ROMPART 9/June
ROMPTR 10/June
RPL operating system 7/June
Rutile crystals ..... 47/Feb.

S

Satellite module rack ..... 7/Oct.
Scan table ..... 20/Oct.
Screen cookbook ..... 31/Oct.
SCPI ..... 16/Apr.
SCPI driver ..... 81/Feb.
SCSI-2 (Small Computer System Interface), Vectra 486 ..... 73/Oct.
Security, Vectra 486 ..... 87/Oct.
Self-test design, sweeper ..... 17/Apr.
Sequences, HP-SL ..... 28/Dec.
Serial distribution network (SDN) . 11/Oct.
Shadowing, Vectra 486 ..... 88/Oct.
Signal processing, HP 11847A .... 74/Apr.
SIMM (single in-line memory module) ..... 78/Oct.
Simulation tool ..... 30/Oct.
Slave module ..... 61/Dec.
Slotline-to-microstrip transition ... 32/Apr.
SoftBench interface, HP Branch Validator ..... 89/Apr.
Software architecture, patient monitor ..... 13/Oct.
Software configuration management 77, 79, 84/June
Software defects ..... 55, 58, 91/Oct.
Software defect costs ..... 55/Oct.
Software defect profit loss calculation ..... 57/Oct.
Software development environment ..... 84/June, 15/Oct.
Software integration, HP Sockets ..... 6/Dec.
Software lifecycle ..... 24/Dec.
Software metrics ..... 55, 58/Oct.
Software performance tools ... 65, 70/June
Solve, HP 48SX 22/June
Source, lightwave ..... 7, 15, 29, 73/Feb.
Source match, changing ..... 28/Apr.
Source, millimeter-wave ..... 50/Apr.
Source, spectrum analyzer 52/June
Source temperature control ..... 16/Feb.
Spectrum analyzer, 150-MHz 44/June
Split-band amplifier ..... 52/Feb.
SS#7 ..... 59/Dec.
ST segments ..... 46/Dec.
Standard display ..... 34/Oct.
Standard parameter interface ..... 17, 19/Oct.
Startup and shutdown, HP Sockets ..... 14/Dec.
State histories ..... 41/Dec.
State specifications ..... 38/Dec.
Strife testing, display ..... 13/Apr.
Stroke index (SI) ..... 41/Oct.
Structure chart ..... 53/Dec.
Structured analysis ..... 53/Dec.
Structured design ..... 53/Dec.
Sweep dynamics, spectrum analyzer 55/June
Sweepers, to 50 GHz ..... 6/Apr.
Symbolic identification ..... 16/Oct.
Syntax checker ..... 39/Oct.
System administration 93/June
System integrator, HP Sockets ..... 8/Dec.
System invariants, HP-SL ..... 37/Dec.
Systemic vascular resistance (SVR) ..... 41/Oct.

T

TAB ICs 41/June
Tagged queuing ..... 70/Oct.
Task windows ..... 35/Oct.
Telephone network monitoring system ..... 59/Dec.
Test automation, HP Branch Validator ..... 83/Apr.
Test environment, automated ..... 49/Oct.
Test opportunities, HP Branch Validator ..... 83/Apr.
Test sets, RF and lightwave ... 13, 20/Feb.
Testing, software ..... 83/Apr., 91/Oct.
Topcase assembly, HP 48SX 26/June
Transimpedance amplifier ..... 65/Feb.
Translation tool ..... 40/Oct.
Transmission measurements, lightwave ..... 11/Feb.
Types, HP-SL ..... 27/Dec.

U

Unequally spaced diodes ..... 34/Apr.
Usability tests ..... 31/Oct.
User interface compiler 57/June
User interface, lightwave analyzer 19/Feb.
User interface, patient monitor ... 29/Oct.
User interface, sweeper ..... 12/Apr.

V

Values, HP-SL ..... 27/Dec.
Variable speed control ..... 85/Oct.
Ventricular stroke work (LVSW/RVSW) ..... 41/Oct.
Version control 77/June
Virtual processor ..... 15/Oct.
Vision, automated 43/June
Visual shell (vsh) ..... 89/Feb.

W

Walk-off ..... 46/Feb.

Part 3: Product Index

Domain software engineering environment (DSEE) ..... June
HP 11846A π/4 DQPSK I-Q generator ..... Apr.
HP 11847A π/4 DQPSK modulation measurement software .... Apr.
HP 11982A lightwave converter ..... Feb.
HP 33324M attenuator ..... Apr.
HP 33326M attenuator ..... Apr.
HP 33327M attenuator ..... Apr.
HP 3588A spectrum analyzer ..... June
HP 48SX scientific expandable calculator ..... June
HP 81210LI isolator ..... Feb.
HP 81310LI isolator ..... Feb.
HP 81520A detector ..... Feb.
HP 81521B detector ..... Feb.
HP 8153A lightwave multimeter ..... Feb.
HP 81530A power sensor ..... Feb.
HP 81531A power sensor ..... Feb.
HP 81532A power sensor ..... Feb.
HP 81536A power sensor ..... Feb.
HP 81533A head adapter ..... Feb.
HP 81551MM laser source ..... Feb.
HP 81552SM laser source ..... Feb.
HP 81553SM laser source ..... Feb.
HP 81554SM laser source ..... Feb.
HP 83420A lightwave test set ..... Feb.
HP 83421A lightwave source ..... Feb.
HP 83422A lightwave modulator ..... Feb.
HP 83423A lightwave receiver ..... Feb.
HP 83424A lightwave CW source ..... Feb.
HP 83425A lightwave CW source ..... Feb.
HP 83557A millimeter-wave source module ..... Apr.
HP 83558A millimeter-wave source module ..... Apr.
HP 8360 Series synthesized sweepers ..... Apr.
HP 83620A synthesized sweeper ..... Apr.
HP 83621A synthesized sweeper ..... Apr.
HP 83622A synthesized sweeper ..... Apr.
HP 83623A synthesized sweeper ..... Apr.
HP 83624A synthesized sweeper ..... Apr.
HP 83630A synthesized sweeper ..... Apr.
HP 83631A synthesized sweeper ..... Apr.
HP 83640A synthesized sweeper ..... Apr.
HP 83642A synthesized sweeper ..... Apr.
HP 83650A synthesized sweeper ..... Apr.
HP 83651A synthesized sweeper ..... Apr.
HP 83810A lightwave signal analyzer ..... Feb.
HP 8703A lightwave component analyzer ..... Feb.
HP Branch Validator ..... Apr.
HP Component Monitoring System ..... Oct.
HP E3500A network monitoring system ..... Dec.
HP GlancePlus/UX ..... June
HP GlancePlus/V ..... June
HP GlancePlus/XL ..... June
HP Sockets Gateway ..... Dec.
HP Software Integration Sockets ..... Dec.
HP Vectra 486/25T ..... Oct.
HP Vectra 486/33T ..... Oct.
HP VUE 1.0 ..... Feb.

Part 4: Author Index

Amino, Mitchell J. ..... Dec.
Anderson, Giovonnae F. ..... Apr.
Backman, Rex A. ..... June
Baker, Glen M. ..... Apr.
Bank, Leslie ..... Oct.
Bartholomew, Christopher A. ..... Oct.
Bear, Stephen P. ..... Dec.
Beers, Ted W. ..... June
Birgenheier, Raymond A. ..... Apr.
Bischof, Richard S. ..... Apr.
Blakely, Frank W. ..... Oct.
Blanc, Ronald C. ..... Apr.
Bledsoe, J. Daren ..... Dec.
Blevins, David W. ..... Oct.
Boles, Mark E. ..... Oct.
Bossaller, James E. ..... Apr.
Brown, Preston D. ..... June
Byrne, Diana K. ..... June
Carlson, Kirsten C. ..... June
Carmichael, William P. ..... June
Cauthorn, James H. ..... June
Champine, Mark A. ..... Feb.
Chang, Kok-Wai ..... Feb.
Chou, Harry ..... Feb.
Collins, Roberto A. ..... Feb.
Conrad, Geraldine A. ..... Feb.
Cyrus, Judith L. ..... Dec.
Daetz, Douglas. ..... June
Daumüller, Karl ..... Oct.
Davidson, Mark N. ..... Apr.
Dearden, Lon ..... Apr.
DeBello, Nicola ..... Dec.
Derocher, Michael ..... June
Dickie, James P. ..... June
Dowden, Tony ..... Oct.
Dunsmore, Joel P. ..... Feb.
Dupre, Jack ..... Feb.
Eisenstein, Gabe L. ..... June
Flachsländer, Erwin ..... Oct.
Freeman, Curtis W. ..... Dec.
Fulton, Kathleen A. ..... Dec.
Garcia, Philip ..... Oct.
Givens, Cynthia ..... Dec.
Goldsack, Patrick G. ..... Dec.
Goodnow, John W. ..... June
Göring, Dieter ..... Oct.
Graf, John D. ..... Oct.
Graves, Spencer B. ..... June
Grossbach, Wolfgang ..... Oct.
Grosthor, Christophe ..... Oct.
Gulland, Scott A. ..... Dec.
Haag, Lance E. ..... Apr.
Harkins, Daniel R. ..... Feb.
Harper, Patrick B. ..... Apr.
Harper, Steven L. ..... June
Harry, Paul D. ..... Dec.
Hart, Michael G. ..... Feb.
Heim, Werner E. ..... Oct.
Heinzelman, Michael A. ..... Feb.
Hentschel, Christian ..... Feb.
Hernday, Paul R. ..... Feb.
Hillstrom, Timothy L. ..... June
Hoover, David M. ..... Apr.
Ikemoto, Mark ..... Dec.
Jahn, Robert ..... Feb.
Jerbic, Mike ..... Oct.
Johnson, Paul ..... Oct.
Jones, Irvin R., Jr. ..... Oct.
Jones, Robert W. ..... June
Jungerman, Roger L. ..... Feb.
Kaiser, Winfried ..... Oct.
Koenig, Mary K. ..... Apr.
Ladeau, B. Robert ..... Dec.
Lang, Marilyn J. ..... Oct.
Lindberg, Thomas B. ..... June
Linsky, Mark ..... Oct.
Lubkin, David C. ..... June
Lum, Gary W. ..... Oct.
Maier, Frank A. ..... Feb.
Maisenbacher, Bernd ..... Feb.
Marciulionis, Roy M. ..... Apr.
Mason, Roy L. ..... June
Mazzucato, Giuseppe ..... Dec.
McQuate, David J. ..... Feb.
Megowan, Patrick J. ..... June
Meier, Wilhelm ..... Oct.
Miller, Christopher M. ..... Feb.
Miranda Alan C. ..... Dec.
Moore, Lester S. ..... June
Müller, Emmerich ..... Feb.
Muranami, M. Jack ..... June
Murray, Bryan P. ..... June
Narayanan, Viswanathan S. ..... Oct.
Neuder, David L. ..... Apr.
Newton, Steven A. ..... Feb.
Oblad, Roger P. ..... Apr.
Patton, Charles M. ..... June
Pless, Wilfried ..... Feb.
Posenato, Antonio ..... Dec.
Pott, Michael ..... Feb.
Rawson, Rollin F. ..... Feb.
Raynham, Michael B. ..... Oct.
Regazzi, John R. ..... Apr.
Reiche, Martin ..... Oct.
Reichert, Wolfgang ..... Feb.
Richardson, Ronald F. ..... June
Riper, Richard W. ..... June
Rivoir, Jochen ..... Feb.
Rochlitzer, Frank ..... Oct.
Rometsch, Rainer ..... Oct.
Rush, Tony W. ..... Dec.
Sayed, Mohamed M. ..... Apr.
Schmidt, Siegmar ..... Feb.
Schuster, Otto ..... Oct.
Schweikardt, Horst ..... Feb.
Seibel, Michael J. ..... Apr.
Shintaku, Larry ..... Oct.
Silvestri, Marco ..... Dec.
Sorin, Wayne V. ..... Feb.
Smith, David L. ..... June
Smith, Irene S. ..... Dec.
Smith, Mark A. ..... June
Smith, Mark M. ..... June
Stead, James R. ..... Apr.
Tarantino, Joseph F. ..... June
Thom, Douglas M. ..... Oct.
Thomas, Joe ..... June
Tivig, Gerhard ..... Oct.
Tom, Thomas ..... Oct.
Vallelunga, John V. ..... Feb.
Veteran, David R. ..... Apr.
Vogel, Eric L. ..... June
Ward, William T. ..... Oct.
Wardle, Jay M. ..... June
Weisner, Steven J. ..... Oct.
Weller, Joachim ..... Oct.
Westerteicher, Christoph ..... Oct.
Wickes, William C. ..... June
Wicklund, Eric J. ..... June
Williams, David A. ..... Feb
Wilson, Edith. ..... June
Wong, Roger W. ..... Feb.
Worsley, Robert S. ..... June
Yarnell, Jimmie L. ..... Feb.
Zellers, James R. ..... Apr.

Hewlett-Packard Company, P.O. Box 51827
Palo Alto, CA 94303-0724

ADDRESS CORRECTION REQUESTED

Bulk Rate
U.S. Postage
Paid
Hewlett-Packard
Company

HEWLETT-PACKARD
JOURNAL

December 1991 Volume 42 • Number 5

Technical Information from the Laboratories of Hewlett-Packard Company

Hewlett-Packard Company, P.O. Box 51827
Palo Alto, California, 94303-0724 U.S.A.

Yokogawa-Hewlett-Packard Ltd., Suginami-Ku Tokyo 168 Japan

Hewlett-Packard (Canada) Ltd.
6877 Goreway Drive, Mississauga, Ontario L4V 1M8 Canada

00093254
MR. DEAN A. LAMPMAN
5 HICKORY WOOD
LOVELAND OH 45140-9424

CHANGE OF ADDRESS:

To subscribe, change your address, or delete your name from our mailing list, send your request to Hewlett-Packard Journal, P.O. Box 51827, Palo Alto, CA 94303-0724 U.S.A. Include your old address label, if any. Allow 60 days.

5091-3007E